VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Návrh a implementace datové struktury Trie

Design and implementation of Trie data structure

2018 Bc. Michal Smolka

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Michal Smolka**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Návrh a implementace datové struktury Trie
Design and Implementation of Trie Data Structure

Jazyk vypracování: čeština

Zásady pro vypracování:

Trie (také prefixový strom) je datová struktura sloužící k uložení řetězců. Jednotlivé implementace uzlů trie a časové složitosti vyhledávání znaků řetězců v uzlech se liší v závislosti na doméně řetězců a velikosti uzlů. Cílem této práce je nastudovat, naimplementovat a otestovat datovou strukturu trie.

Diplomant musí v rámci diplomové práce:
1. Nastudovat datovou strukturu trie a její varianty.
2. Naimplementovat a otestovat datovou strukturu trie a její varianty.
3. Začlenit implementaci do stávajícího katedrálního frameworku.
4. Porovnat výkon implementace s výkonem jiných datových struktur.

Seznam doporučené odborné literatury:

[1] H. Samet. Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann, 2006.
[2] Arnab Bhattacharya. Fundamentals of Database Indexing and Searching (1st ed.). Chapman & Hall/CRC, 2014.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Peter Chovanec, Ph.D.**

Datum zadání: 01.09.2017
Datum odevzdání: 30.04.2018

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA * Fakulta elektrotechniky a informatiky * OSTRAVA * 1D16

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2018 .....

Chtěl bych touto cestou poděkovat vedoucímu této práce Ing. Petru Chovanci, Ph.D. za jeho odbornou pomoc při sepisování této práce. Dále bych chtěl poděkovat doc. Ing. Michalovi Krátkému, Ph.D. za jeho pomoc a nápady při implementaci datové struktury Trie a její následné optimalizaci.

Abstrakt

Tato diplomová práce se zabývá popisem a implementací datové struktury Trie. Je zde podrobně popsána datová struktura Trie, včetně operací nad touto datovou strukturou. Jsou zde zmíněny výhody a nevýhody této struktury, a také je zde navrženo řešení pro potlačení některých nedostatků datové struktury Trie. Následně tato diplomová práce popisuje datové struktury vycházející z Trie, včetně její persistentní varianty. V další kapitole je pak vysvětleno stránkování datových struktur a jeho účel. V implementační části je popsán databázový framework RadegastDB a jeho využití při implementaci Trie. Je zde také popsána implementace Trie v jazyce C++, včetně její optimalizované verze. Následuje výkonnostní otestování této struktury a jejich variant na několika datových kolekcích. A jejich následné porovnání s B^+-stromem a sekvenčním polem, implementovaných v rámci frameworku RadegastDB.

Klíčová slova: Trie, stránkování, RadegastDB, datová struktura, komprese řetězců

Abstract

This diploma thesis deals with the description and implementation of Trie data structure. The Trie data structure is described in detail, including operations above this data structure. There are mentioned the advantages and disadvantages of this structure, and there is also a solution designed to overcome some deficiencies in the Trie data structure. Subsequently this diploma thesis describes data structures based on Trie, including its persistent variant. The next chapter explains the paging of data structures and their purpose. The implementation part describes the RadegastDB database framework and its use in Trie implementation. There is also described the implementation of Trie in C ++, including its optimized version. This is followed by performance testing of this structure and its variants on several data collections. And their subsequent comparison with B^+-tree and sequential array, implemented within RadegastDB framework.

Key Words: Trie, paging, RadegastDB, data structure, strings compression

Obsah

Seznam použitých zkratek a symbolů

DB	–	Databáze
SŘBD	–	Systém pro řízení báze dat
DBMS	–	Database management system
XML	–	eXtensible Markup Language
RAM	–	Random access memory
CPU	–	Central processing unit
LRU	–	Last recently used
FIFO	–	First in-First out
LIFO	–	Last in-First out
HDD	–	Hard disk drive
ASCII	–	American Standard Code for Information Interchange

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Dnešní době, tedy začátku 21. století se občas také říká Informační věk. Je to z toho důvodu, že informace se staly nedílnou součásti každodenního všedního života. Informace jsou jednoduše dostupné téměř pro každého, kdo má přístup k internetu, tedy ke globální mezisíti, sdílející a zpřístupňující informace dostupné téměř odkudkoliv. Nic z toho by však nebylo možné, kdyby neexistoval způsob ukládání informací na nějáka obrovská a rychlá datová úložiště. Mluvíme-li o datech, jedná se o informace převedené do jazyka zpracovatelného počítačem. Elementárně se jedná o tok bytů či bitů, které jsou počítačem zpracovávány.

K uchovávání dat slouží databázové systémy, které zahrnují softwarové prostředky pro modifikaci a přístup k těmto datům. Takovým softwarovým prostředkům se v češtině říká, systém pro řízení báze dat, zkráceně SŘBD. V anglické literatuře se pak setkáváme s pojmem database management system, zkráceně DBMS. Tyto SŘBD se vzájemně často liší, ovšem každý z nich zpřístupňuje data pomocí datových struktur. Jedná se o způsob organizace dat v paměti počítače, který umožňuje, aby data byla zpracovávána efektivně.

Jedním z konkurentů mezi SŘBD, se stává databázový framework RadegastDB[7], jedná se o framework vyvíjený již několik let databázovou skupinou na Katedře informatiky, Fakulty elektrotechniky a informatiky, Vysoké školy Báňské - Technické univerzity Ostrava. Cílem RadegastDB je poskytnout maximální výkon místo implementace bohaté funkcionality, jako je podpora metod uložených v databázi atd.[7]. Jako každý SŘBD, i tento využívá k efektivní manipulaci s daty datové struktury. Existuje několik typů databázových systémů, některé pracují pouze s hlavní paměti počítače, tedy s paměti RAM, jiné pouze se sekundární pamětí, což bývá nejčastěji HDD. Výhodou využití SŘBD využívající sekundární paměť je trvalost dat uložených v této paměti, jejich nevýhodou je ovšem dlouhá přístupová doba. SŘBD využívající pouze hlavní paměť bývají několikanásobně rychlejší, problém je v tom, že ztratí-li paměť RAM napájení, je-li tedy počítač vypnut, pak ztratíme veškerá data uložená v tomto typu paměti. Nejvhodnějším řešením se tedy jeví využít jakýsi kompromis mezi těmito přístupy. Nabízí se tedy možnost ukládat data do sekundární paměti, přičemž je jejich část načtená v hlavní paměti. Mezi SŘBD využívající tento přístup patří i RadegastDB. Takové SŘBD ukládají data do tzv. stránek, tyto stránky jsou uloženy na sekundárním úložišti a některé z nich v hlavní paměti, ve frontě stránek označované jako cache buffer.

Existují tedy datové struktury uložené v hlavní paměti, označované za in-memory struktury. Pak jsou zde struktury využívající stránkování, takové datové struktury jsou persistentní neboli trvalé, protože jsou uloženy na sekundárním úložišti. Datové struktury se obvykle vyvíjejí za účelem rychlého přístupu k datům. Takovým strukturám se říká vyhledávací struktury a jako nejefektivnější se mezi nimi jeví stromové struktury z teorie grafů. Takové datové struktury mají svůj kořen, tedy počáteční uzel, dále vnitřní uzly a koncové uzly, kterým se říká listové. V dnešní době existuje celá řada vyhledávacích stromových struktur a každá je vhodná pro jiný účel.

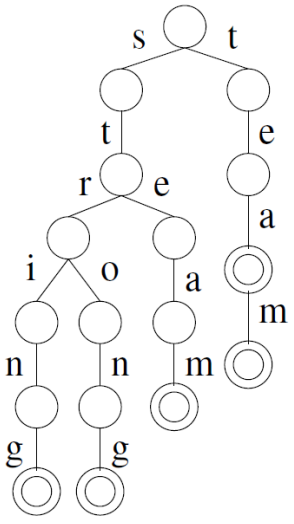
Chceme-li data vyhledávat pomocí nějakých klíčových slov sdružovaných do řetězců, využi-

jeme datovou strukturu určenou k vyhledávání řetězců. Mezi vhodné kandidáty se řadí datové struktury vycházející z Trie[2, 3]. Cílem této diplomové práce je tedy naimplementovat tuto datovou struktury v jazyce *C++*, zakomponovat ji do databázového frameworku RadegastDB, čímž zajistíme její persistenci a porovnat tuto implementaci s jinými datovými strukturami implementovanými v rámci frameworku RadegastDB, pomocí výkonnostních testů. Základní verze této struktury, je kvůli některým svým vlastnostem nevhodná k širšímu využití. Tato práce si dále klade za cíl optimalizovat tuto strukturu a potlačit některé její nedostatky.

2 Datová struktura Trie

2.1 Popis datové struktury

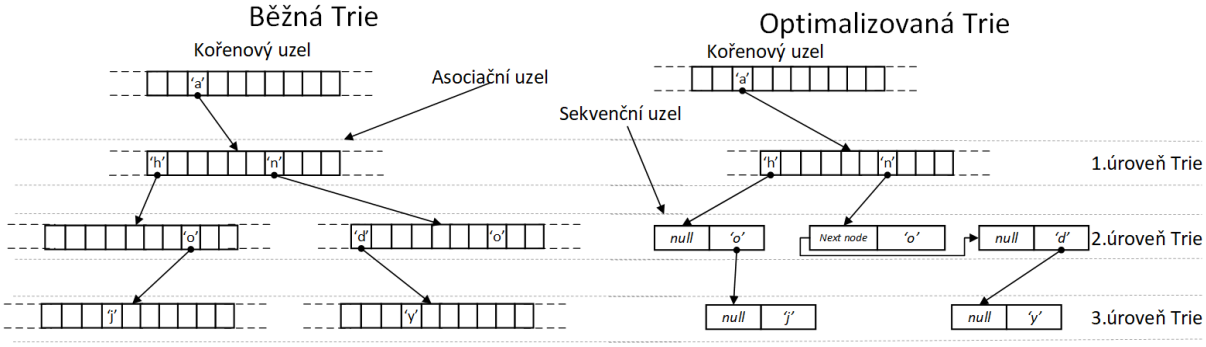
Trie[2, 3] je paměťová stromová datová struktura, která je díky svému přístupovému času vhodná pro indexaci řetězců. Název 'Trie' je odvozen od anglického slova 'retrieval', což znamená 'vyhledávání'[1]. Každá cesta stromem od kořene k listu představuje řetězec, přičemž každý vnitřní uzel představuje prefix tohoto řetězce. Je běžné, že různé řetězce obsahují stejný prefix, to je důvodem proč se Trie říká také prefixový strom[1]. Trie se často využívá jako přístupová struktura pro vyhledávání dvojic klíč - hodnota, kde klíč představuje řetězec uložený v Trie a hodnota libovolné dodatečné informace, na které se odkazuje listový uzel Trie. Na Trie lze nahlížet jako na konečný deterministický automat, kde kořenový uzel představuje počáteční stav a listové uzly představují konečné nebo přijímací stavy. Obrázek 1 zobrazuje Trie obsahující řetězce(klíče) 'string', 'strong', 'steam', 'tea' a 'team'. Každá hrana struktury představuje znak řetězce v Trie. Obsahuje-li Trie řetězce různých délek se společnými prefixy, je nutné přidávat speciální znak na konec každého řetězce, aby bylo možné rozpoznat listový uzel. Struktura vnitřního uzlu se oproti listovému liší pouze v tom, že listový uzel obsahuje již zmíněný speciální znak, a také se odkazuje na hodnotu zmíněné dvojice klíč-hodnota. Vnitřní uzly obsahují pouze odkaz na svého potomka. Na obrázku 1 ze zdroje [4] jsou to řetězce 'tea' a 'team', které obsahují společný prefix, konec řetězce je zde identifikován dvojitým kruhem, který rovněž označuje listový uzel. Trie snižuje redundanci ukládání všech řetězců, nicméně pokud jsou dva řetězce podobné, ale liší se pouze v prvním znaku, jsou uloženy podél různých cest Trie. Řetězce 'steam' a 'team', jsou příkladem extrémního případu, kdy jsou podobné řetězce uloženy zcela samostatně, i když jeden řetězec je příponou druhého[4].



Obrázek 1: Základní Trie

2.1.1 Typy uzlů Trie

Základní Trie, tedy výše popsaná Trie, využívá tzv. asociační uzly. Tento typ uzlů umožňuje díky své velikosti přímé adresování potomků těchto uzlů, adresa potomka se nachází na pozici v uzlu, která odpovídá konkrétní hodnotě řetězce uloženého v Trie. Nevýhodou tohoto typy uzlů je však to, že dochází k plýtvání místem, protože velikost tohoto typu uzlů je statická. Vhodným řešením je použití řetězce uzlů, kde každý uzel obsahuje uloženou hodnotu, odkaz na potomka a odkaz na následující uzel[5]. V následujících kapitolách budu o tomto typu uzlu mluvit jako o sekvenčním. Jak už pojmenování uzlu napovídá, prvky v těchto uzlech jsou ukládány sekvenčně a neplýtvá se tedy tolik místem v uzlu. Nevýhodou tohoto typu uzlu je, že může docházet k dlouhému sekvenčnímu prohledávání těchto uzlů, čímž se degraduje časová složitost operací téměř na lineární. Abychom předešli tomuto problému, využívají se tyto uzly v nižších úrovních Trie (dále od kořenového uzlu), kde počet prvků v těchto uzlech je nízký a prohledávání těchto uzlů se týká jen několika málo prvků, obvykle pouze jednoho. Trie využívající sekvenční uzly bude v následujících kapitolách označována jako optimalizovaná Trie. Na obrázku 2 je srovnání běžné Trie využívající pouze asociační uzly a optimalizované Trie využívající sekvenční uzly od 2. úrovně.



Obrázek 2: Srovnání uzlů Trie

Pro srovnání běžné a optimalizované Trie 2, jsem zvolil jiného zobrazení uzlů datové struktury, pro lepší pochopení reálné implementace. Uzly již nepředstavují stavy deterministického konečného automatu, ale paměťová pole. V obou typech Trie, jsou vloženy řetězce 'ano', 'ahoj' a 'andy'. Doména těchto řetězců je osmi-bitová, tedy velikost asociačního uzlu je 256 B, nebo také 256 znaků. Znaky jsou na obrázku umístěny v asociačních uzlech většinou nahodile, nicméně skutečnou pozici těchto znaků lze vyjádřit pomocí tabulky ASCII. Sekvenční uzly obsahují pouze jednu položku, nicméně lze zvolit i větší velikost těchto uzlů. Uzly na 2. úrovni Trie jsou pro prefix 'an' zřetězeny pomocí ukazatele (*next node*), sekvenční uzly bez ukazatele na následující uzel obsahují hodnotu *null*.

2.1.2 Prostorová složitost typů uzlů Trie

Asociační uzly mají pro osmi-bitovou doménu velikost 2^8 bytů, lze ovšem využít i jiné velikosti domén. Dále mějme doménu o velikosti *d*. Díky této velikosti uzlů lze v tomto typu uzlů využít přímé adresování prvku Trie, protože každou doménu řetězce lze převést na *d*-bitové číslo, které určuje pozici prvku v uzlu, nicméně nevýhodou tohoto typu uzlů je jeho prostorová složitost, protože v nižších úrovních Trie jsou tyto uzly téměř prázdné. Nemají-li vkládané řetězce společné prefixy, potom je pro každý vkládaný řetězec délky *n*, vytvořeno *n-1* nových asociačních uzlů, což je při *d*-bitové doméně $2^d \times (n-1)$ nových bytů. Tento typ uzlů je tedy vhodný pouze pro uzly blízké kořenovému uzlu, kde lze předpokládat jejich vyšší využití. Prostorová složitost základní Trie je tedy dána počtem uzlů Trie *n* a velikostí domény vkládaných záznamů v bitech *d*, celková prostorová složitost je tedy $2^d \times n$. Tato složitost je však pouze teoretická neboť v různých implementacích potřebujeme, kromě uložené domény také další data, jako ukazatele na uzel potomka. U implementace Trie, je nutné používat 2 ukazatele, protože prvek v uzlu Trie musí kromě odkazu na potomka, mít možnost uložit také odkaz na hodnotu z již zmíněné dvojice klíč-hodnota. Mějme tedy velikost ukazatele *v*, pak má Trie reálnou velikost

$$2 \times v \times 2^d \times n \text{ [B]}$$

Tento nedostatek je důvodem k optimalizaci základní Trie a využití sekvenčních uzlů.

Velikost sekvenčního uzlu pro *d*-bitovou doménu je *d* bitů. Optimalizovaná Trie využívající pouze sekvenční uzly by pak měla teoretickou velikost $n \times d$ bitů, kde *n* je počet uzlů. Reálně však sekvenční uzly Trie obsahují více položek, aby se částečně zamezilo dynamické alokaci paměti. Mějme tedy sekvenční uzel určený pro *k* prvků. Sekvenční uzel musí dále obsahovat informaci o počtu prvků které obsahuje, tato hodnota má velikost nejčastěji 2 B, může být ovšem i jiná, označme ji *m*. Dále musí prvek sekvenčního uzlu obsahovat opět 2 ukazatele o velikosti *v*, jednoho na potomka a druhého na hodnotu dvojice klíč-hodnota. Pak je tedy reálná velikost optimalizované Trie využívající pouze sekvenční uzly

$$n \times (m + k \times (2 \times v + \frac{d}{8})) \text{ [B]}$$

Mějme tedy například Trie s osmi-bitivou doménou z obrázku 2, velikostí ukazatele 4 B a u optimalizované Trie sekvenční uzly určené pro 1 prvek a velikost hodnoty uchovávající počet prvků v uzlu 2 B. Velikost běžné Trie by pak byla

$$(2 \times 4 \times 2^8) \times 6 = 12288 \text{ B}$$

a velikost optimalizované Trie by byla

$$(2 \times 4 \times 2^8) \times 2 + 5 \times (2 + 1 \times (2 \times 4 + \frac{8}{8})) = 4151 \text{ B}$$

2.2 Operace nad základní Trie

Všechny operace prováděné nad základní Trie mají asymptotickou časovou složitost *O(k)*, kde k je délka vkládaného, hledaného či mazaného řetězce.

2.2.1 Vkládání řetězce do základní Trie

Proces vkládání začíná jako u všech stromových struktur u kořene. Označíme-li kořen jako uzel na n-té úrovni, kde n je rovna 0, postupujeme následovně.

1. Zjišťujeme, zda z uzlu U, tedy uzlu n-té úrovně vystupuje hrana ohodnocena znakem na *(n+1)*. pozici vkládaného řetězce. Existuje-li taková hrana pokračujeme krokem 3, neexistuje-li pokračujeme krokem 2.

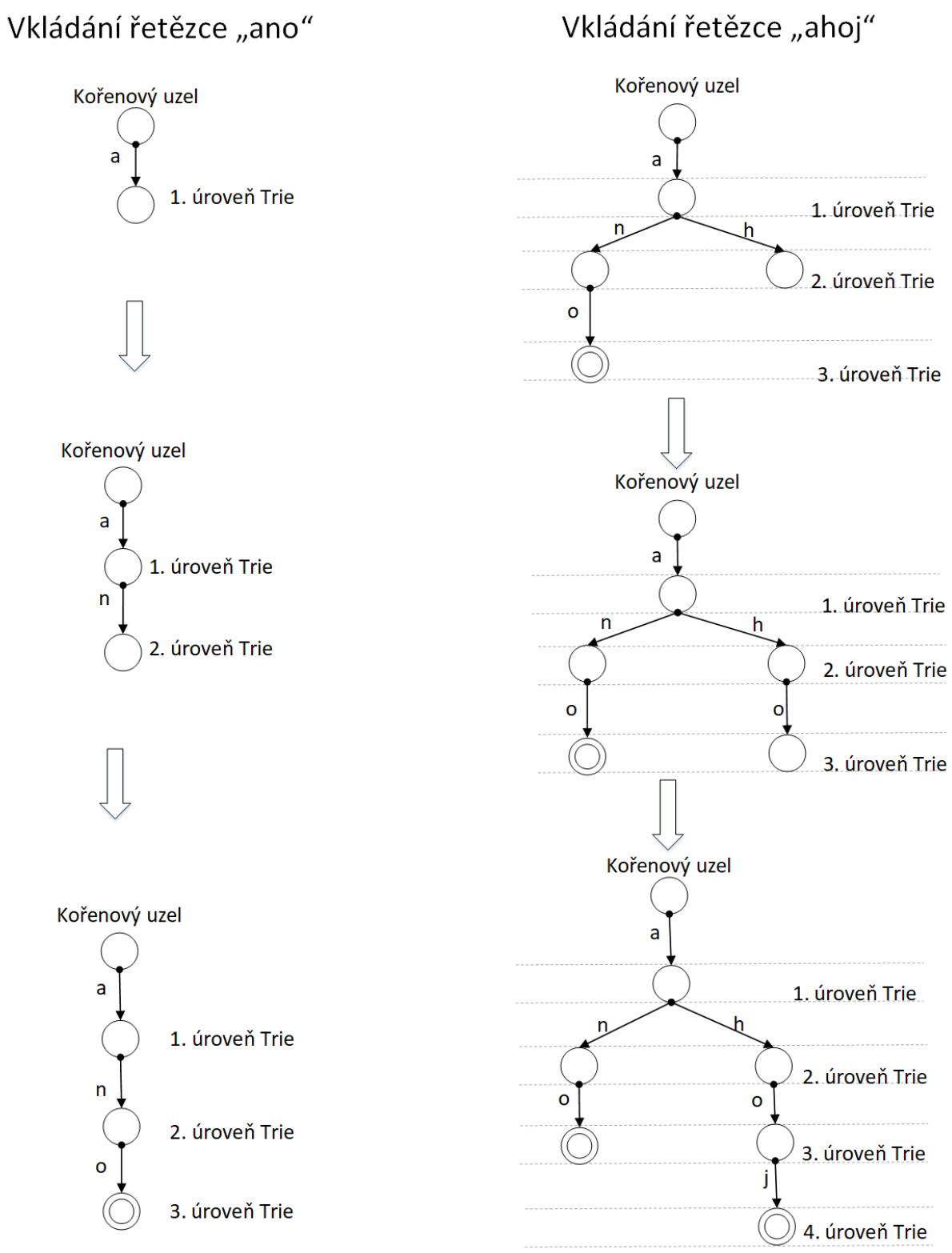
2. Vytvoříme uzel V na *(n+1)*. úrovni Trie a hranu s ohodnocením *(n+1)*-tého znaku vkládaného řetězce, vedoucí z uzlu U do uzlu V.

3. Je-li *n+1* menší než délka vkládaného řetězce, zvýšíme n o 1 a pokračujeme krokem 1. Je-li *n+1* rovno délce vkládaného řetězce, pokračujeme krokem 4.

4. Označíme uzel V jako listový a vkládání je dokončeno.

Obrázek 3 zobrazuje vložení řetězců 'ano' a 'ahoj' do Trie a postup je následující. Po vytvoření Trie existuje pouze kořenový uzel označující prázdný řetězec. Vložení řetězce 'ano' do prázdné Trie začíná vytvořením nového uzlu na 1. úrovni Trie, k tomuto uzlu vede hrana z kořene s ohodnocením 'a', která reprezentuje 1. znak řetězce 'ano'. V dalším kroce neboli v další iteraci tohoto cyklu je vytvořen nový uzel na 2. úrovni Trie a hrana ohodnocena znakem 'n' vedoucí do nového uzlu. V posledním kroce je vytvořen uzel na 3. úrovni Trie a hrana ohodnocena 3. znakem vkládaného řetězce, tedy znakem 'o', znak 'o' je posledním znakem vkládaného řetězce a nový uzel je tedy označen za listový (označeno dvojitým kruhem).

Vložení řetězce 'ahoj' do Trie obsahující řetězec 'ano' je následující. Z kořenového uzlu již vede hrana ohodnocena 1. znakem vkládaného řetězce, překročíme tedy k další iteraci tohoto cyklu. Z uzlu 1. úrovně, jehož vstupní hrana je ohodnocena znakem 'a', nevystupuje hrana ohodnocena 2. znakem vkládaného řetězce, je tedy nutné vytvořit nový uzel na 2. úrovni Trie a k němu odpovídající hranu. Tento postup se opakuje, dokud není vložen poslední znak vkládaného řetězce a uzel k němuž směřuje hrana ohodnocena tímto znakem je označen jako listový.



Obrázek 3: Vložení řetězců "ano"a "ahoj"do Trie

2.2.2 Hledání řetězce v základní Trie

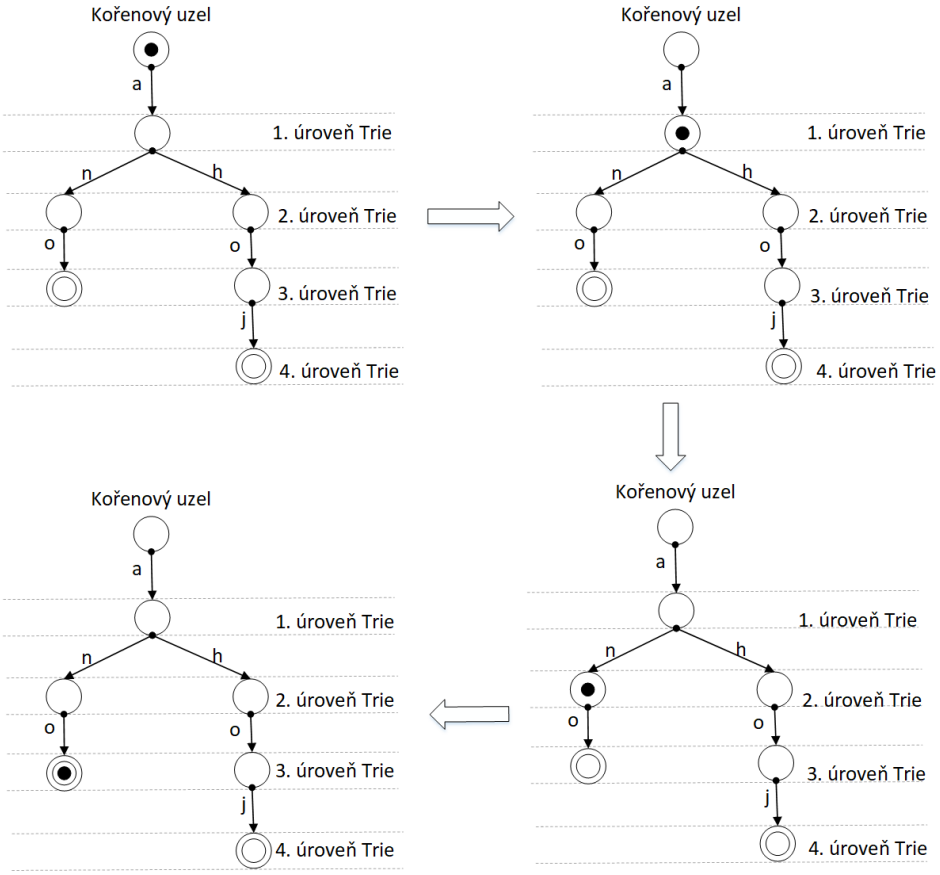
Při hledání se opět začíná u kořenového uzlu, který označíme jako kontextový, zavedeme si proměnnou n rovnu 1, která označuje n-tý znak hledaného řetězce. Postup je následující.

1. Zjišťujeme, zda z kontextového uzlu vystupuje hrana ohodnocena n-tým znakem hledaného řetězce. Pokud ano pokračujeme dále, pokud ne hledání končí a řetězec nebyl nalezen.

2. Přesouváme se po hraně ohodnocené n-tým znakem hledaného řetězce do uzlu další úrovně, který si označíme jako kontextový.

3. Je-li n rovno délce hledaného řetězce a současně je kontextový uzel listovým uzlem, pak je hledání ukončeno a řetězec byl nalezen. Je-li n rovno délce hledaného řetězce, ale kontextový uzel není listový, hledání končí neúspěchem. Je-li n menší, než délka hledaného řetězce zvýšíme jej o 1 a pokračujeme krokem 1.

Tento postup je zobrazen na obrázku 4, kde probíhá vyhledávání řetězce 'ano' v Trie obsahující řetězce 'ano' a 'ahoj', kontextový uzel je v každé iteraci algoritmu označen černým puntíkem uvnitř uzlu.



Obrázek 4: Hledání řetězce "ano"v Trie

Hledání začíná v kořenovém uzlu, kde zjišťujeme, zda z něj vede hrana ohodnocena 1. znakem hledaného řetězce, tedy 'a'. Tato hrana existuje, a proto se přesuneme po této hraně do uzlu následující úrovně Trie. Opět zjišťujeme, zda z tohoto uzlu vystupuje hrana ohodnocena znakem 'n', tato hrana existuje, a proto se po ní přesouváme do uzlu v další úrovni Trie. Nyní hledáme hranu s ohodnocením 'o', která rovněž existuje. Přesouváme se tedy po ní do další úrovně Trie, zde zjišťujeme, že hledaný řetězec neobsahuje žádné další znaky a současně je uzel v němž se nacházíme listový. Hledání tedy končí úspěchem, protože Trie obsahuje celý hledaný řetězec.

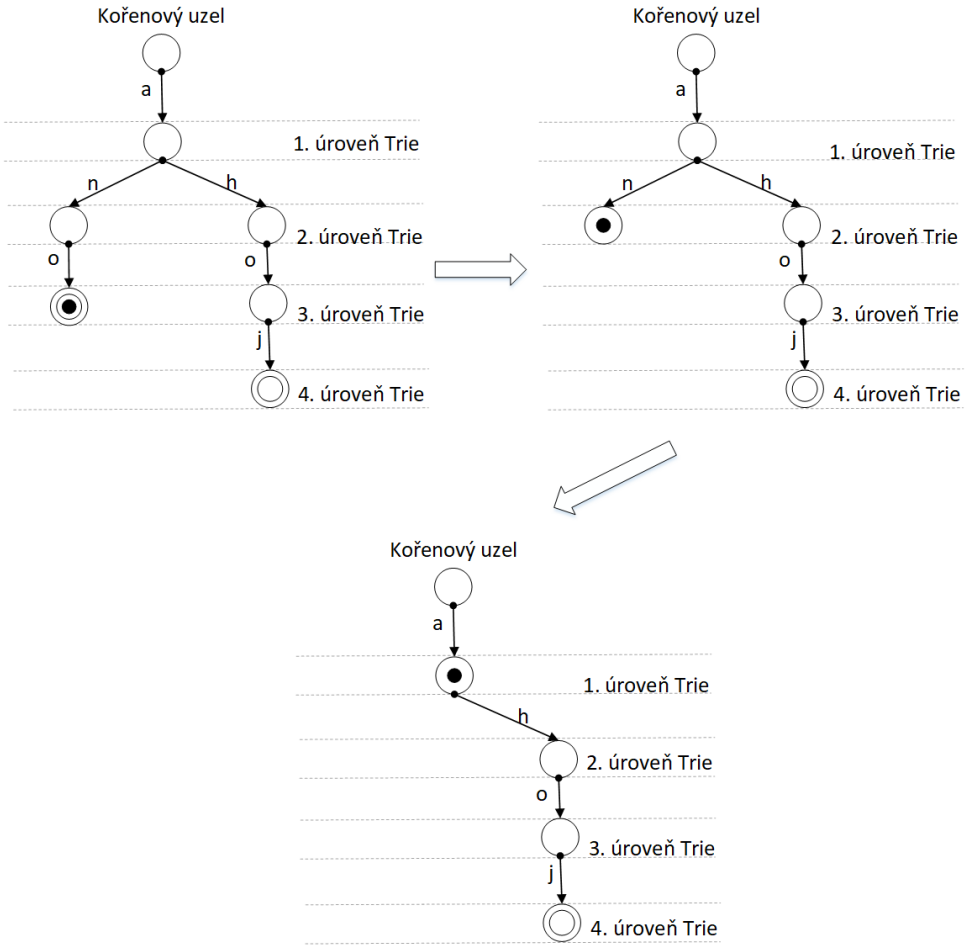
2.2.3 Mazání řetězce ze základní Trie

Mazání řetězce z Trie začíná opět od kořene a má následující postup.

1. Vyhledání listového uzlu pro daný řetězec viz 2.2.2.
   - Řetězec byl nalezen, listový uzel označíme jako kontextový.
   - Řetězec nebyl nalezen a proces mazání končí neúspěchem.
2. Zjistíme počet hran vystupující z kontextového uzlu.
   - Je-li počet hran větší než 0 a současně je kontextový uzel v n-té úrovni, kde n je délka mazaného řetězce, smažeme indikátor listového uzlu, uzel se stane vnitřním a proces mazání tím úspěšně končí.
   - Je-li počet hran větší, než 0 a současně není kontextový uzel v n-té úrovni, kde n je délka mazaného řetězce, pak je mazání úspěšně dokončeno.
   - Počet hran je 0, kontextový uzel je listový a není v n-té úrovni, kde n je délka mazaného řetězce, pak proces mazání končí úspěchem.
   - Počet hran je 0, pokračujeme dalším krokem.
3. Smažeme tento uzel, včetně jeho vstupní hrany a rodič tohoto uzlu se stává kontextovým.
4. Není-li kontextový uzel současně kořenovým uzlem, pokračujeme krokem 2.

Tento proces uvažujme na následujícím příkladu. Mějme Trie obsahující řetězce 'ano' a 'ahoj' z obrázku 4 a řetězec ke smazání 'ano'. Tento postup je zobrazen na obrázku 5, kontextový uzel je označen černým puntíkem uvnitř uzlu.

Nejprve je nalezen listový uzel odpovídající hledanému řetězci. Tento uzel včetně jeho vstupní hrany je smazán, protože počet jeho výstupních hran je roven 0. Kontextovým uzlem se stává předek smazaného uzlu ležící v 2. úrovni Trie. Počet výstupních hran kontextového uzlu je roven 0, a proto bude tento uzel opět smazán včetně jeho vstupní hrany. Kontextovým uzlem se nyní stává jeho předek na 1. úrovni Trie. Tento uzel má 1 výstupní hranu a současně se úroveň tohoto uzlu neshoduje s délkou řetězce, mazání řetězce 'ano' je tedy úspěšně dokončeno.



Obrázek 5: Mazání řetězce "ano"z Trie

2.2.4 Aktualizace základní Trie

Existují dva typy aktualizace Trie, aktualizace hodnoty a aktualizace klíče. Proces aktualizace hodnoty je využíván pro změnu hodnoty daného klíče. Jedná se o vyhledávání v Trie viz 2.2.2 a následné přepsání hodnoty, na kterou se listový uzel daného klíče odkazuje. Proces aktualizace klíče se skládá ze dvou kroků, a to ze smazání původního klíče viz 2.2.3 a následného vložení nového klíče viz 2.2.1 do Trie. Mají-li starý a nový klíč společný prefix délky n, je maximální počet smazaných uzlů roven m-n, kde m je délka starého klíče. Mějme tedy délku společného prefixu k. Proces aktualizace klíče probíhá následovně.

1. Vyhledání listového uzlu pro řetězec určený k aktualizaci viz 2.2.2.
   - Řetězec byl nalezen, listový uzel označíme jako kontextový.
   - Řetězec nebyl nalezen a proces aktualizace končí neúspěchem.
2. Je-li k rovno úrovni kontextového uzlu, pokračujeme krokem 6.

3. Zjistíme počet hran vystupujících z kontextového uzlu.

    - Je-li počet hran větší než 0 a současně je kontextový uzel v n-té úrovni, kde n je délka starého řetězce, smažeme indikátor listového uzlu, uzel se stane vnitřním a pokračujeme krokem 8.
    - Je-li počet hran větší, než 0 a současně není kontextový uzel v n-té úrovni, kde n je délka starého řetězce, pak pokračujeme krokem 6.
    - Počet hran je 0, kontextový uzel je listový a není v n-té úrovni, kde n je délka starého řetězce, pak pokračujeme krokem 6.
    - Počet hran je 0, pokračujeme dalším krokem.

4. Smažeme tento uzel včetně jeho vstupní hrany a rodič tohoto uzlu se stává kontextovým.

5. Není-li kontextový uzel současně kořenovým uzlem, pokračujeme krokem 2.

6. Je-li kontextový uzel listovým uzlem a současně je úroveň kontextového uzlu rovna délce starého řetězce, pokračujeme krokem 7, jinak krokem 8.

7. Odstraníme indikátor listového uzlu z kontextového uzlu a uzel se stává vnitřním.

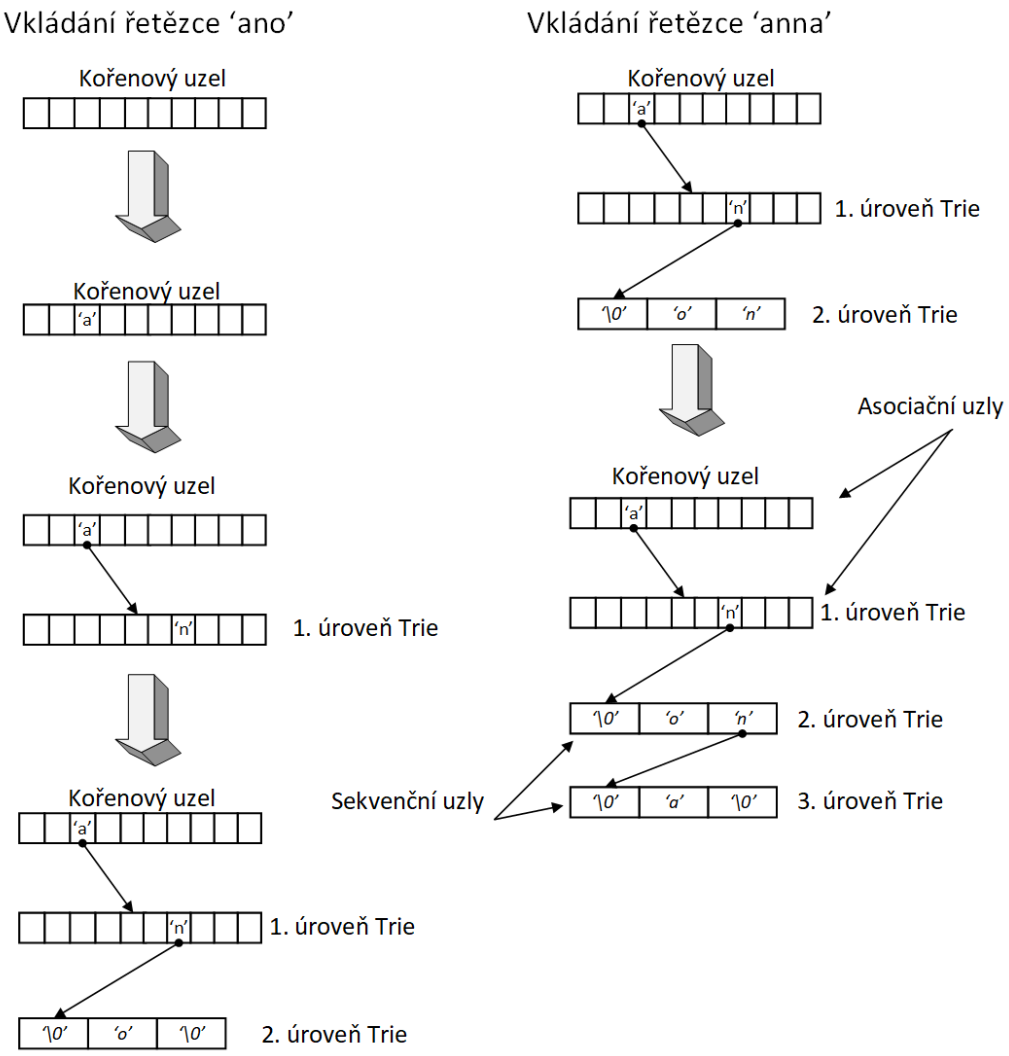
8. Pokračujeme algoritmem vložení řetězce viz 2.2.1

Tento postup do značné míry kopíruje algoritmus mazání řetězce z Trie viz 2.2.3. K účelu aktualizace klíče v Trie lze použít i naivní postup, kdy je starý klíč smazán a nový je vložen, nicméně algoritmus popsaný výše je efektivnější, protože bere v potaz délku společného prefixu, a tím snižuje počet kroků nutných k aktualizaci klíče Trie. Při naivním postupu, je počet smazaných a vytvořených uzlů v nejhorším případě $m+n$, kde m je délka starého klíče a n délka nového, zatímco u výše popsaného algoritmu je to $m + n - 2 \times k$, kde k je délka společného prefixu.

2.3 Operace nad optimalizovanou Trie

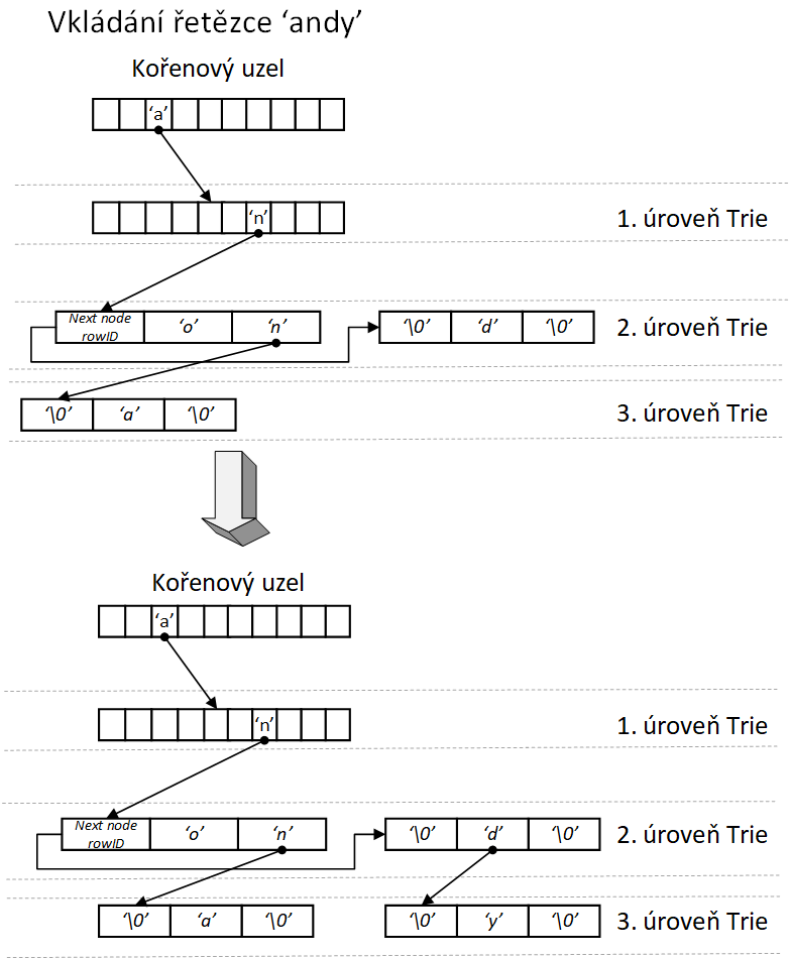
2.3.1 Vkládání do optimalizované Trie

Vkládáni u Trie využívající sekvenční uzly začíná stejným způsobem jako do základní Trie, dokud nedochází ke vkládání do sekvenčního uzlu. Vkládání do sekvenčního uzlu začíná od určené úrovně Trie, tedy od x - tého znaku vkládaného řetězce. Pokud již existuje sekvenční uzel, do kterého vkládáme, musíme jej celý sekvenčně projít abychom zamezili vkládání duplicit. Pokud se znak v tomto uzlu nenachází vložíme jej na poslední volné místo v sekvenční uzlu. Pokud sekvenční uzel neexistuje, vytvoříme jej a vložíme x - tý znak řetězce na začátek tohoto uzlu. Je-li sekvenční uzel plný, vytvoříme nový sekvenční uzel a tyto uzly zřetězíme, x - tý znak poté vložíme na začátek nového sekvenčního uzlu. Od x - té úrovně se pak tento postup opakuje, dokud není vložen poslední znak vkládaného řetězce. Na obrázku 6 je vidět vkládání řetězců 'ano' a 'anna' do optimalizované Trie.



Obrázek 6: Vložení řetězce "ano"a řetězce "anna"

Tato Trie obsahuje sekvenční uzly od druhé úrovně, tzn. od třetího znaku řetězce. Vkládání řetězce ’ano’ je pro první dvě iterace algoritmu vkládání 2.2.1 stejné, jako u základní verze Trie. Velikost sekvenčního uzlu je zde stanovena na dvě položky. U třetí iterace se znak ’o’ vloží na začátek nově vytvořeného sekvenčního uzlu. Vkládání řetězce ’anna’ začíná modifikovat Trie od třetí iterace algoritmu vkládání, kdy je vložen znak ’n’ za znak ’o’ do sekvenčního uzlu. Předešlé iterace vkládání tohoto řetězce pouze sestupovaly na další úrovně Trie, protože se dané prvky již v uzlech nacházejí. Ve čtvrté iteraci algoritmu je vytvořen nový sekvenční uzel a znak ’a’ je vložen na jeho začátek. Sekvenční uzly obsahují na začátku znak ’\0’, který identifikuje prázdné paměťové místo, nachází-li se tento znak na první pozici sekvenčního uzlu, pak tento uzel prozatím neobsahuje ukazatele na následující sekvenční uzel. Je-li zmíněný znak jinde, pak značí prázdnou pozici pro prvek v sekvenčním uzlu. Následující obrázek 7 ukazuje co se stane, když sekvenční uzel přeteče.



Obrázek 7: Vložení řetězce "andy"

Při vkládání řetězce ’andy’, proběhlo během prvních dvou iterací algoritmu vkládání pouze sestupování do dalších úrovní Trie, protože se první dva znaky vkládaného řetězce v Trie již vyskytují. Během třetí iterace algoritmu vkládání jsme zjistili, že sekvenční uzel, do kterého chceme vložit znak ’d’ je již plný a současně se v něm ještě tento znak nenachází. Je tedy nutné vytvořit nový sekvenční uzel a ukazatel na tento uzel uložit do plného uzlu, ukazatel v obrázku označuje box ’next node rowID’. Znak ’d’ se tedy uloží na začátek nového uzlu a pokračuje se obvyklým způsobem.

2.3.2 Vyhledávání v optimalizované Trie

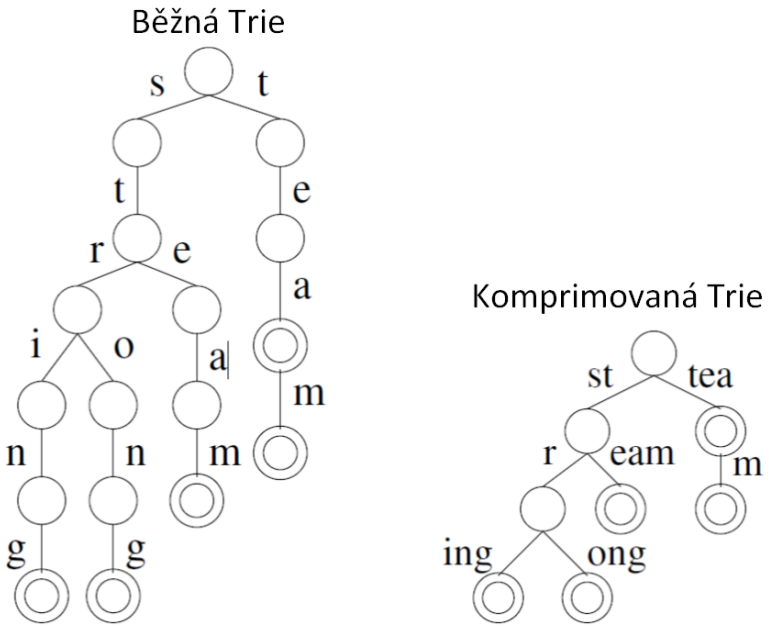
Hledání u optimalizované Trie začíná stejným způsobem jako u základní Trie, dokud nedochází k hledání v sekvenčním uzlu. Tento uzel se musí sekvenčně projít, dokud není nalezen hledaný znak nebo se tento znak v uzlu nevyskytuje a hledání končí neúspěchem. Je-li znak nalezen pokračuje se opět prohledáváním uzlu na které se předešlý znak odkazoval. Tento postup se opakuje, dokud není nalezen poslední znak řetězce.

Mějme tedy například Trie z obrázku 7 obsahující řetězce 'ano', 'anna' a 'andy'. Chceme-li vyhledat řetězec 'andy' postupujeme následovně. Nejprve přistoupíme v kořenovém uzlu na pozici znaku 'a' a zjišťujeme, zda obsahuje ukazatel na následující uzel, tento prvek se odkazuje na následující uzel, a proto sestoupíme přes ukazatele do další úrovně Trie. Zde opět přistoupíme na pozici znaku 'n', opět se zde nachází ukazatel, po němž sestupujeme na další úroveň Trie. Následuje přístup k sekvenčnímu uzlu, je tedy nutné jej sekvenčně procházet a porovnávat jednotlivé prvky tohoto uzly s hledaným znakem 'd'. V prvním sekvenčním uzlu se tento znak nenachází, tento uzel však obsahuje ukazatele na následující sekvenční uzel, v tomto uzlu se znak 'd' již nachází a sestupujeme tedy přes jeho ukazatel na 3. úroveň Trie. Zde se znak 'y' nachází hned na první pozici, hledaný řetězec neobsahuje další znaky a prvek s označením 'y' v tomto uzlu je listový. Hledání tedy končí úspěšně.

2.4 Varianty Trie

Existuje řada variant datové struktury Trie, které odstraňují některé její nedostatky nebo z této struktury vyplývají. Některé varianty se zaměřují na kompresi této struktury a jiné na její persistenci

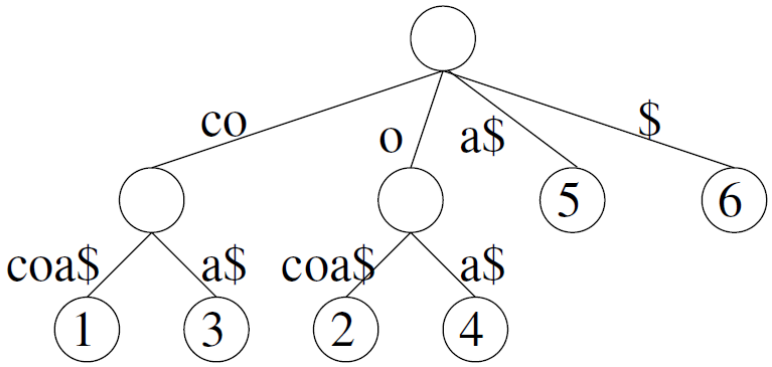
Prostorově úsporná varianta Trie, nazývaná komprimovaná Trie (někdy také kompaktní Trie), komprimuje všechny unární uzly, tedy uzly s pouze jedním potomkem, do jednoho uzlu[10]. Tato varianta Trie je ovšem vhodná pouze pro statické implementace, protože operace modifikující tuto strukturu jsou složité a časově náročné. Na obrázku 8 ze zdroje [4] je srovnání běžné a komprimované Trie.



Obrázek 8: Srovnání běžné a komprimované Trie

Na obrázku 8 ze zdroje [4] je vidět, že uzly s pouze jedním potomkem byly sloučeny do jednoho, čímž se snížila výška stromu. Nelze ovšem sloučit všechny uzly s jedním potomkem do jednoho, protože bychom ztratili některé uložené řetězce, např. řetězec 'team' nemůže být s kořenového uzlu odkázán ihned, protože bychom tím ztratili řetězec 'tea'.

PAT-strom[14] neboli sufixový strom je datová struktura ukládající všechny podřetězce vloženého řetězce. Tato datová struktura vychází z komprimované Trie, které se někdy přezdívá i PATRICIA (Practical Algorithm To Retrieve Information Coded In Alphanumeric), odtud tedy název PAT-strom. PAT-strom je struktura vhodná pro ukládání signatur, a hlavně pro fulltextové vyhledávání, protože umožňuje nalézt všechny podřetězce daného řetězce. Tato varianta je zobrazena na obrázku 9 ze zdroje [4], tento PAT-strom byl sestaven pro řetězec 'cocoa'. Znak '$' je znakem ukončující daný podřetězec.



Obrázek 9: PAT-strom

Když se objem dat stává příliš vysokým, je většina dat uložena na sekundárním úložišti (obvykle HDD), spíše než v hlavní paměti[6].

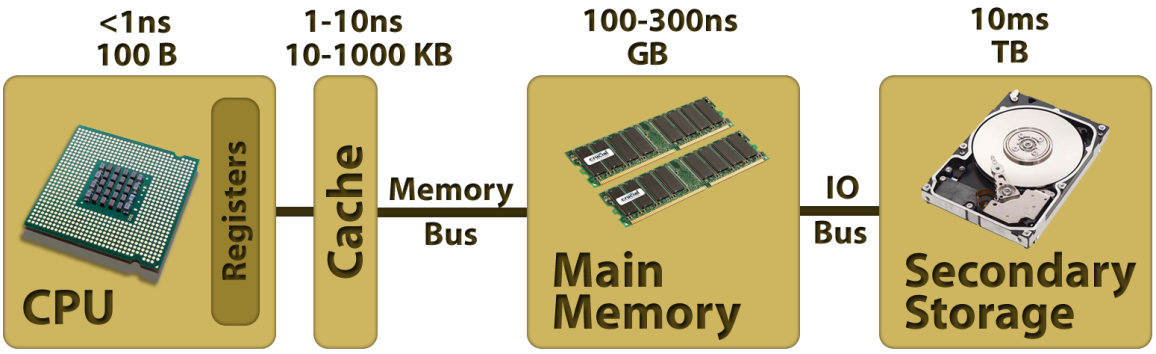
Persistentní variantou Trie je TPT (Tightly Packaged Trie)[13]. Jedná se o kompaktní Trie vhodnou spíše pro čtení, protože vychází z komprimované Trie a umožňuje rychlé stránkování s krátkými časy čtení. Je vhodná především pro ukládání n-gramů a frázových tabulek pro statistický strojový překlad.

Tato diplomová práce se zabývá implementací persistentní Trie, tedy implementací, která umožňuje ukládat datovou strukturu na sekundární úložiště. V předchozích kapitolách byla popsána paměťová Trie, tedy datová struktura, která je uložena v hlavní paměti RAM. Problémem je, že při každém následování ukazatele na uzel, může dojít u persistentní Trie k přístupu na disk, to je mnohem dražší operace, než adresování v rámci hlavní paměti[6]. Jak jsem se již zmínil struktura Trie je prostorové velmi náročná, oproti jiným datovým strukturám obsahuje mnohem větší počet uzlů, což vede k častějšímu naplnění mezipaměti tzv. cache bufferu. Vysoký počet uzlů tedy vede také k vyššímu počtu tzv. logických přístupů, což jsou operace čtení nebo zápisu z cache bufferu nebo sekundárního úložiště. To ovšem není jediný problém, protože velikost cache bufferu není neomezená. Dosáhne-li cache buffer naplnění a je vyžadován zápis

dalšího uzlu, musí se provést tzv. fyzický přístup, což je přístup k sekundárnímu úložišti. Tento přístup je časově velmi náročný a vede tedy ke zpomalení operací prováděných nad datovou strukturou. Dalším problémem je, že nelze využít paměťové ukazatele, protože hlavní paměť je během operací nad cache bufferem přepisována a docházelo by k narušení datové struktury. Jednotlivé uzly datové struktury tedy neobsahují ukazatele do hlavní paměti, ale identifikátory stránek datového souboru umístěného na sekundárním úložišti, čímž docílíme správné identifikace jednotlivých stránek datového souboru. Uzly Trie mohou být ovšem menší, než stránky datového souboru a aby nedocházelo k plýtvání místem, musejí se uzly Trie sekvenčně uložit do stránky. Z tohoto důvodu, musí být vedle identifikátoru stránky datového souboru, uložena další hodnota určující pořadí uzlu ve stránce.

3 Stránkování datových struktur

S rostoucím množstvím dat nastává problém s kapacitou datových úložišť. Často není možné udržovat celou datovou strukturu v hlavní paměti RAM, a proto se využívá technika zvaná stránkování, která ukládá datovou strukturu na sekundární úložiště a část této struktury udržuje v hlavní paměti. Většina databázových systémů podporuje zpracovávaní transakcí a s nimi související infrastrukturu. Tyto databázové systémy využívají různé stránkované datové struktury ke zpracování dat[8]. Tato diplomová práce se zaměřuje na stránkovanou Trie. Stránkovaná struktura je persistentní a její data jsou uloženy do stránek neboli bloků. Stránka je primárně uložena v sekundární paměti. Proto se její velikost volí obvykle jako násobek velikosti sektoru disku (obvykle 512 B) a velikosti jednotky souborového systému (obvykle 2 kB). Typická velikost stránky je tedy 8 nebo 16 kB. Na obrázku 10 ze zdroje [8] je paměťová hierarchie reprezentována CPU registry nahoře a diskem dole, včetně jejich propustností a kapacit.

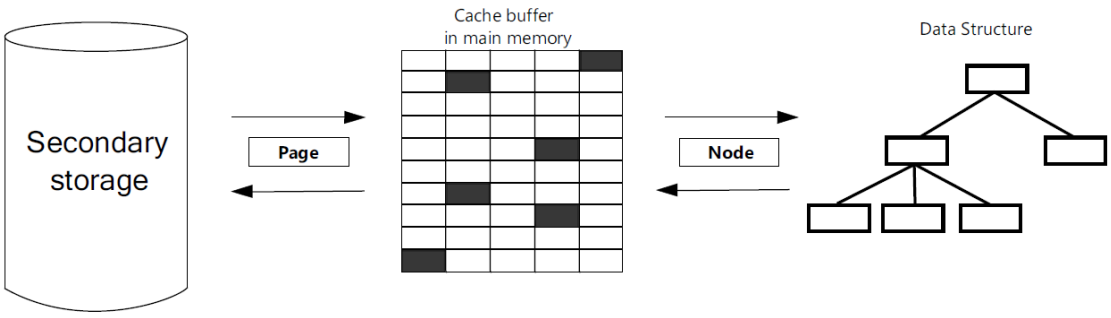


Obrázek 10: Paměťová hierarchie

Z obrázku je zřejmé, že hlavní paměť má až 100 000x vyšší propustnost než sekundární paměť. Pro práci s datovými strukturami musí být uzel datové struktury uložen v hlavní paměti počítače, tedy v paměti RAM[4]. Během zpracovávání jsou tato data přenesena do mezipaměti, tedy do paměti cache, odkud jsou pomocí CPU a jeho registrů dále zpracovávána. S rostoucí vzdáleností zpracovávaných dat od CPU, roste i doba k jejich zpracování. Cílem je tuto vzdálenost zkracovat na minimum. Využíváme zde techniku, které se někdy říká optimalizace pro L2 cache. Tato technika zajišťuje, že aktuálně zpracovávaná data jsou uložena v mezipaměti počítače a jsou tedy mnohem rychleji zpracována. Databázové systémy ovšem pracují s velkým množstvím dat, není tedy možné, aby všechna data byla uložena v hlavní paměti a už vůbec ne v paměti cache, protože čím je paměť rychlejší a blíže k CPU, tím je její kapacita menší a náklady na pořízení vyšší. Toto je důvodem pro využívání persistentních datových struktur, které jsou uloženy v sekundárním úložišti a jejichž část je uložena v hlavní paměti

Každý databázový systém přiděluje část hlavní paměti, která se nazývá cache buffer[8]. Cache buffer je fronta stránek uložena v hlavní paměti[9, 11]. Pokud datová struktura vyžaduje

stránku, nejdříve se zkontroluje, zda se nachází v cache bufferu, pokud ano načtou se tato data do mezipaměti, tedy do paměti cache, pokud ne jsou tato data načtena ze sekundárního úložiště do cache bufferu a následně do mezipaměti. Tento princip je zobrazen na obrázku 11 ze zdroje [8]



Obrázek 11: Stránkovaná datová struktura s cache bufferem

Rozlišujeme dva typy přístupu ke stránce v cache bufferu. Fyzický přístup načítá stránku ze sekundárního úložiště, zatím co logický přístup je každý přístup ke stránce nehledě na to, zda je už v cache bufferu, nebo je nutné načíst ji ze sekundárního úložiště. Je-li stránka nalezena v cache bufferu hovoříme o tzv. cache hit, není-li nalezena hovoříme o cache miss[8]. Efektivita cache bufferu je měřena frekvencí cache hitů, tzv. cache hit rate. Je vyjádřena vzorcem *(cache hits / logické přístupy) * 100 [%]*. V případě, že jsou všechny stránky načteny v cache bufferu je cache hit rate 100%.

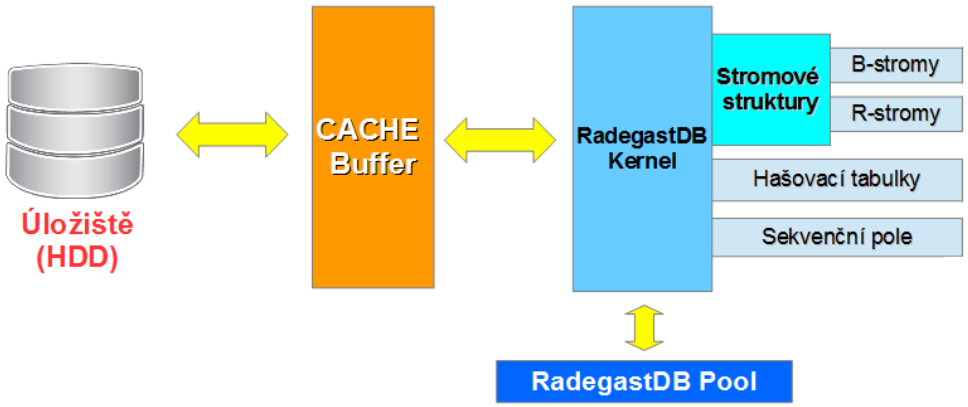
Algoritmy přístupu k datům využívají inteligentní techniky pro ukládání dat do cache bufferu. Jedním z nejpoužívanějších algoritmů je algoritmus LRU. Jedná se o značkovací algoritmus, tedy každé zpracovávané stránce se přiřadí značka, která určuje využití stránky. Stránka s nejvyšším počtem využití má nejvyšší pravděpodobnost, že bude v budoucnu opět využita. Pro načtení nové stránky do cache bufferu je tedy vybrána stránka s nejmenším počtem použití a ta je zapsána na sekundární úložiště. Volné místo v cache bufferu potom obsadí požadovaná stránka ze sekundárního úložiště. Existuje řada jiných algoritmů pro ukládání stránek na sekundární úložiště, jedním z nich je například optimalizovaná verze algoritmu LRU, která se nazývá LRU-K[12]. Některé jednoduché cache buffery využívají taky algoritmy jako FIFO, LIFO nebo podobné.

4 Implementace Trie

Cílem této práce je navrhnou a implementovat datovou strukturu Trie v jazyce *C++* a začlenit tuto implementaci do databázového frameworku RadegastDB[7]. První důležitá otázka, kterou si bylo nutné položit byla, zda budeme implementovat paměťovou nebo persistentní Trie. Tato práce se zabývá implementací persistentní Trie, což sebou nese oproti paměťové implementaci několik problému. Prvním problémem bylo, že velikost uzlu Trie se liší od velikosti datového bloku v cache bufferu. Bylo tedy nutné ukládat uzly Trie sekvenčně do datového bloku. Dalším problémem byl obrovský počet logických a fyzických přístupů u základní Trie, protože asociační uzly Trie jsou jen velmi řídce využívány u nižších úrovní Trie (dále od kořene Trie) a dochází tak k plýtvání místem v uzlech Trie, tedy i v datových blocích. Toto neodvratně vede k vysokému množství datových bloků v cache bufferu, a tedy i časově náročnějšímu vyhledávání jednotlivých datových bloků v cache bufferu. Řešením bylo využití optimalizované Trie, která tento problém minimalizovala, protože procentuální využití jejich uzlů je mnohem vyšší a nedochází k vytváření obrovského množství datových bloků. V následujících podkapitolách popíši implementaci konkrétních částí datové struktury Trie a její začlenění do databázového frameworku RadegastDB.

4.1 Databázový framework RadegastDB

RadegastDB[7] je databázový framework vyvíjený již několik let databázovou skupinou na Katedře informatiky, Fakulty elektrotechniky a informatiky, Vysoké školy Báňské - Technické univerzity Ostrava. Cílem RadegastDB je poskytnout maximální výkon místo implementace bohaté funkcionality, jako je podpora metod uložených v databázi atd.[7]. Tento framework je psán v programovacím jazyce *C++* a umožňuje testovat různé datové struktury. V průběhu vývoje došlo k přejmenování frameworku z QuickDB na RadegastDB, ale názvy tříd zůstaly nezměněny. Na následujícím obrázku 12 je model architektury frameworku RadegastDB.



Obrázek 12: Model architektury frameworku RadegastDB

4.1.1 Cache buffer

Základní strukturu frameworku tvoří třída *cQuickDB*, která uchovává instance tříd *cNodeCache*, *cMemoryPool* a stará se o stránkování struktur. Instance této třídy je použita při vytváření datových struktur.

Třída *cNodeCache* umožnuje načítání jednotlivých uzlů datových struktur do operační paměti, což zrychluje práci s nimi. Všechny operace s datovými strukturami jsou prováděny v rámci operační paměti. Tato třída dále zajišťuje zamykání načtených uzlů, aby byla zajištěna konzistence datových struktur.

Třída *cMemoryPool* slouží jako pomocná paměť při provádění operací na datovými strukturami. Tuto třídu představuje na obrázku 12 RadegastDB Pool. Tato třída je stejně jako *cNodeCache* sdílená pro celý framework.

Třída *cQuickDB* obsahuje tři základní metody. Metoda *Create* slouží k vytvoření persistentního souboru v sekundární paměti, vytváří instanci třídy *cNodeCache* a je nezbytná pro přípravu k použití. Metoda *Open* slouží k otevření persistentního souboru. Obě tyto metody přijímají stejné parametry. Prvním parametrem je jméno databáze, které je využito při pojmenování persistentního souboru. Druhým parametrem je velikost přidělené paměti cache, třetím je maximální velikost uzlu v paměti a posledním parametrem je persistentní velikost jednoho bloku. Ke smazání slouží metoda *Close*, která přijímá jako parametr booleovskou hodnotu, určující, zda budou uzly z paměti odstraněny.

4.1.2 Datové typy

Implementace Trie podporuje systém datových typů frameworku RadegastDB. Systém datových typů slouží k reprezentaci záznamů a je implementován třídami *cTuple* a *cNTuple*, které reprezentují konkrétní záznamy. Třída *cTuple* reprezentuje záznam pevné délky s homogenní doménou. Obsahuje pole položek stejných datových typů a reprezentuje tedy n-dimenzionální n-tici. Třída *cNTuple* reprezentuje homogenní záznam s proměnlivou délkou. Oba tyto datové typy obsahují řadu metod umožňujících jejich čtení, modifikaci, kopírování, porovnávání, kódování a spoustu dalších metod. Datové *cTuple* a *cNTuple* jsou reprezentovány polem s homogenními datovými typy, jako celočíselné typy a typy s pohyblivou řadovou čárkou. Tyto typy využívají třídu *cSpaceDescriptor*, která obsahuje metadata o konkrétních typech záznamů.

4.1.3 Využití frameworku RadegastDB

V této práci je využita struktura frameworku RadegastDB, určená pro stránkování datových struktur a systém datových typů. Pro úspěšné využití frameworku pro stránkování datové struktury, je nutné, aby hlavička datové struktury dědila ze třídy *cDStructHeader*, pomocí níž jsou zaregistrovány jednotlivé typy uzlů datové struktury určené ke stránkování. Tato třída obsahuje virtuální metody *Read* a *Write*, které je nutné naimplementovat a slouží k uložení hlavičky datové struktury na sekundární úložiště. Tyto metody se volají během volání funkce *Close* objektu

třídy *cQuickDB*. Hlavička datové struktury obsahuje metadata a statistické informace o datové struktuře, nutné ke správné funkci datové struktury.

Další třídou určenou pro podporu stránkování datových struktur je třída *cNode*. Z této třídy dědí všechny typy uzlů datové struktury. Tato třída opět obsahuje metody pro čtení a zápis na sekundární úložiště, tedy metody *Read* a *Write*, které jsou volány při volání funkce *WriteNode* a *ReadNode* objektu dědící ze třídy *cNodeHeader*. Tato třída dále obsahuje blok dat *mData* určené velikosti, který představuje uzel datové struktury, popř. několik uzlů. Tento blok je pak pomocí operací *Read* a *Write* zapisován na disk s metadaty uzlu. Další důležitou proměnnou této třídy je *mHeaderID*, které určuje typ uzlu.

Třída *cNodeHeader* určuje typ uzlu a obsahuje čistě virtuální metody *WriteNode* a *ReadNode* sloužící k zápisu nebo čtení konkrétního typu uzlu ze sekundárního úložiště. Tyto metody jsou volány pro každý typ uzlů v metodě *Close* třídy *cQuickDB* nebo při ukládání a čtení bloků ze sekundárního úložiště během zpracovávání operací nad datovými strukturami. Tyto operace jsou řízeny funkcemi *ReadR*, *ReadW* a *ReadNew* třídy *cNodeCache*.

Funkce *ReadR*, *ReadW* a *ReadNew* třídy *cNodeCache* slouží ke čtení, zápisu nebo vytvoření bloku cache bufferu. Funkce *ReadR* a *ReadW* přijímají jako parametry identifikátor uzlu a identifikátor hlavičky uzlu. Pomocí těchto informací je načten konkrétní požadovaný typ uzlu. Tyto operace také vytvářejí čtecí, nebo zapisovací zámky, které je nutné odemknout po skončení práce s uzly pomocí metod *UnlockR* a *UnlockW* jejímž parametrem je instance třídy *cNode*, tedy odemykaný blok konkrétního typu. Metoda *ReadNew*, jak už název napovídá vytvoří nový blok konkrétního typu, zadaného parametrem pomocí identifikátoru hlavičky uzlu a zamkne blok zapisovacím zámkem.

Tyto popsané metody jsou dostačující k funkční implementaci stránkované datové struktury využívající framework RadegastDB.

4.2 Implementace datové struktury

Datová struktura Trie je v mé implementaci reprezentována třídou *cTrie* využívající šablony *TItem*, *TRowID* a *TDomain*. Šablona *TItem* zastupuje typ záznamu ze systému datových typů RadegastDB (viz kapitola 4.1.2). Šablona *TRowID* zastupuje datový typ identifikátoru pozice položky v bloku (viz kapitola 4.2.1). Poslední šablona *TDomain*, zastupuje typ domény zpracovávaných záznamů.

Hlavička datové struktury Trie je reprezentována třídou *cTrieHeader*, která dědí ze třídy *cDStructHeader* popsané v kapitole 4.1.3. Tato třída slouží jako persistentní úložiště pro metadata datové struktury a její statistiky. Statistikami datové struktury Trie jsou její maximální, minimální a průměrná výška. Dále procentuální využití jednotlivých typů uzlů Trie, včetně počtu prvků v těchto uzlech. Dalšími statistikami jsou počet uložených klíčů, počet bloků tvořících datovou strukturu a počet jednotlivých typů uzlů Trie.

Třída *cTrie* poskytuje několik veřejných metod pro práci se strukturou. Jedná se o bezparametrický konstruktor *cTrie*, dále metody *Create* a *Open*, které přijímají jako parametry, instance tříd *cQuickDB* a *cTrieHeader*. Metoda *Create* slouží k vytvoření datové struktury a k registraci hlavičky struktury třídou *cNodeCache* pomocí metody *Register* s parametrem instance třídy *cTrieHeader*. Dále slouží k výpočtu atributů nutných pro správnou práci s identifikátory pozice v bloku. Metoda *Open* se od *Create* liší pouze v tom, že slouží k otevření již existující struktury.

Další metody třídy *cTrie* slouží k modifikaci a čtení z datové struktury. Jedná se o metody *Insert* 1, *PointQuery* 2 a *Update* 3. Metoda *Insert* slouží k vložení záznamu, metoda *PointQuery* k vyhledání hodnoty záznamu dle zadaného klíče a metoda *Update* k aktualizaci hodnoty záznamu pro daný klíč. Začněme popisem metody *Insert*. Metoda je přetížená, a proto budou uvedeny obě varianty ve stejné tabulce 1 oddělené dvojitou horizontální čárou. Návratový typ je *bool*, informuje, zda byl vložen nový klíč, tedy *true*, nebo zda se jedná o duplicitu, tedy *false*.

Název parametru	**Popis parametru**	**Datový typ**
key	reprezentuje klíč záznamu	TItem (cTuple nebo cNTuple)
value	reprezentuje hodnotu záznamu	char*
key	reprezentuje klíč záznamu	char*
length	reprezentuje délku klíče	int
value	reprezentuje hodnotu záznamu	char*

Tabulka 1: Popis parametrů metody Insert

Metoda *Insert* 1 začíná načtením kořenového uzlu do proměnné *contextNode*, která reprezentuje konkrétní uzel pro danou iteraci algoritmu. Identifikátor kořenového uzlu je udržován v hlavičce datové struktury Trie, konkrétně v instanci třídy *cTrieHeader*. Cyklus pro vkládání se opakuje, dokud není vložen poslední znak vkládaného řetězce. Na začátkku cyklu je přečten identifikátor uzlu potomka pro daný klíč, to zajišťuje metoda *GetChildNode*. V dalším kroce algoritmus zjišťuje, zda probíha poslední iterace cyklu, během níž se vkládají data do datového uzlu, v pseudokódu reprezentovaného proměnnou *dataNode*. Proměnná *lastDataNodeRowID* je opět čtena z hlavičky datové struktury, reprezentuje identifikátor uzlu, který ještě není zaplněn, včetně poslední použité pozice v něm. Metoda *GetBlockID* dokáže z proměnné identifikující uzel, vypočítat hodnotu identifikátoru bloku v němž se uzel nachází. Metoda *GetSubBlockID* pak z téže hodnoty vypočítá konkrétní pozici v bloku. Metoda *AddData* vloží data reprezentované proměnnou Value na konkrétní pozici. Metoda *AddItem* vloží konkrétní část klíče na správnou pozici v uzlu, včetně ukazatele na hodnotu v datovém uzlu. Pokud se daný klíč v datové struktuře ještě nenacházel, pak metoda vrací *true*, jinak *false*. Další větev algoritmu je přístupná, pokud je identifikátor potomka roven 0, tedy *childNodeID==0*, tento případ nastane pokud

Algoritmus 1: Metoda Insert

```
Input: Key, Value
Output: true nebo false
Function Insert(Key, Value):
    contextNode ⟵ ReadAsociationNodeW(rootNodeID);
    inserted ⟵ false;
    for index ⟵ 0 to keyLength do
        childNodeID ⟵ contextNode.GetChildNodeID(Key[index]);
        if keyLength == index + 1 then
            dataNode ⟵ ReadDataNodeW(GetBlockID(LastDataNodeRowID));
            dataNode.AddData(GetSubBlockID(LastDataNodeRowID), Value);
            inserted ⟵ contextNode.AddItem(Key[index], LastDataNodeRowID);
        else if childNodeID == 0 then
            childNode ⟵ ReadNewNodeW();
            childNodeID ⟵ childNode.GetIndex()
            contextNode.AddItem(Key[index], childNodeID);
            contextNode ⟵ childNode;
        else
            contextNode ⟵ ReadNodeW(childNodeID);
    end
    UpdateStatisticsInformation(KeyLength, inserted);
    return inserted;
End Function
```

Algoritmus 2: Metoda PointQuery

```
Input: Key
Output: Value
Function PointQuery(Key):
    contextNode ⟵ ReadAsociationNodeW(rootNodeID);
    for index ⟵ 0 to keyLength do
        childNodeID ⟵ contextNode.GetChildNodeID(Key[index]);
        if keyLength == index + 1 then
            dataNodeRowID ⟵ GetRowID(contextNode, Key[index])
            dataNode ⟵ ReadDataNodeR(GetBlockID(dataNodeRowID));
            Value ⟵ dataNode.GetData(GetSubBlockID(dataNodeRowID));
            return Value;
        else if childNodeID == 0 then
            return NULL;
        else
            contextNode ⟵ ReadNodeR(childNodeID);
    end
    return Value;
End Function
```

ještě neexistuje uzel který by danou část klíče obsahoval. V této větvi je tedy načten nový uzel z cache bufferu do proměnné *childNode* a jeho indentifikátor je vložen do proměnné *childNodeID*. Následně je pak konkrétní část klíče vložena do kontextového uzlu, včetně ukazatele na potomka. Pro další iteraci se pak kontextovým uzlem stane uzel potomka. V poslední větvi se pak pouze načte konkrétní uzel potomka do kontextového uzlu, tento případ nastane, pokud potomek již existuje. Před ukončením tohoto algoritmu proběhne aktualizace informací o datové struktuře, jako počet položek a průměrná výška stromu.

Metoda PointQuery 2 je opět přetížená a přijímá následující parametry2.

Název parametru	**Popis parametru**	**Datový typ**
key	reprezentuje klíč záznamu	TItem (cTuple nebo cNTuple)
result	reprezentuje výstupní hodnotu záznamu	&char*
key	reprezentuje klíč záznamu	char*
length	reprezentuje délku klíče záznamu	int
result	reprezentuje výstupní hodnotu záznamu	&char*

Tabulka 2: Popis parametrů metody PointQuery

Metoda *PointQuery* 2 začíná opět načtením kořenového uzlu do proměnné *contextNode*. Následuje cyklus, který končí buď úspěšným nalezením hodnoty hledaného záznamu, nebo jeho nenalezením. V prvním kroce cyklu je opět uložen identifikátor uzlu potomka do proměnné childNodeID. Následuje větvení, kde byl-li klíč nalezen probíhá načtení datového uzlu, odkazovaného z listového uzlu Trie, pro daný klíč. Následně se hledaná hodnota záznamu načte do výstupní proměnné *Value*, pomocí metody *GetData* z dané pozice v uzlu, zjištěné metodou *GetSubBlockID*. Další větev cyklu říká, že pokud je identifikátor uzlu potomka roven 0, pak je hledání neúspěšné a končí. Poslední větev cyklu pouze načítá uzel potomka do proměnné *contextNode*.

Metoda *Update* 3 je také přetížená a přijímá následující parametry 3.

Název parametru	**Popis parametru**	**Datový typ**
key	reprezentuje klíč záznamu k aktualizaci	TItem (cTuple nebo cNTuple)
value	reprezentuje novou hodnotu záznamu	char*
key	reprezentuje klíč záznamu k aktualizaci	char*
length	reprezentuje délku klíče	int
value	reprezentuje novou hodnotu záznamu	char*

Tabulka 3: Popis parametrů metody Update

Metoda *Update* 3 se v principu téměř neliší od metody PointQuery 2. Jediný rozdíl je v první větvi cyklu tohoto algoritmu, kde místo aby se kopírovala hodnota hledaného záznamu do proměnné, tak probíhá opačný proces, při němž je na danou pozici v datovém uzlu zkopírována nová hodnota záznamu. Metoda vrací *true*, pokud byl záznam úspěšně aktualizován a *false* pokud nebyl nalezen.

Algoritmus 3: Metoda Update

Input: $Key, newValue$
Output: **true** nebo **false**
Function `Update(`$Key, newValue$`):`
    $contextNode \longleftarrow ReadAsociationNodeW(rootNodeID)$;
    for $index \longleftarrow 0$ ***to*** $keyLength$ **do**
        $childNodeID \longleftarrow contextNode.GetChildNodeID(Key[index])$;
        if $keyLength == index + 1$ **then**
            $dataNodeRowID \longleftarrow GetRowID(contextNode, Key[index])$
            $dataNode \longleftarrow ReadDataNodeW(GetBlockID(dataNodeRowID))$;
            $dataNode.AddData(GetSubBlockID(dataNodeRowID), newValue)$;
            return true;
        else if $childNodeID == 0$ **then**
            return false;
        else
            $contextNode \longleftarrow ReadNodeR(childNodeID)$;
    end
End Function

Poslední metodou pro modifikaci Trie je metoda *Delete* 4, která maže záznam z Trie. Tato metoda není v této diplomové práci implementována, protože nebyla předmětem zadání. Nicméně je vhodné ukázat postup jakým tato metoda funguje. Metoda *Delete* přijímá následující parametry 4.

Název parametru	Popis parametru	Datový typ
key	reprezentuje klíč záznamu ke smazání	TItem (cTuple nebo cNTuple)
key	reprezentuje klíč záznamu ke smazání	char*
length	reprezentuje délku klíče	int

Tabulka 4: Popis parametrů metody Delete

Metoda *Delete* 4 je opět velmi podobná metodě *PointQuery* 2, s tím rozdílem, že je-li nalezen klíč ke smazání, pak je smazána hodnota záznamu z datového uzlu, na který se odkazuje listový uzel klíče a následně proběhne cyklus, kde postupujeme od listového uzlu směrem ke kořenovému.

Algoritmus 4: Metoda Delete

```
Input: Key
Output: true nebo false
Function Delete(Key):
    contextNode ⟵ ReadAsociationNodeW(rootNodeID);
    for index ⟵ 0 to keyLength do
        childNodeID ⟵ contextNode.GetChildNodeID(Key[index]);
        if keyLength == index + 1 then
            dataNodeRowID ⟵ GetRowID(contextNode, Key[index])
            dataNode ⟵ ReadDataNodeW(GetBlockID(dataNodeRowID));
            dataNode.DeleteData(GetSubBlockID(dataNodeRowID));
            for i ⟵ index to 0 do
                childCount ⟵ contextNode.GetChildCount();
                if (childCount > 0)AND(i == index) then
                    contextNode ⟵ ReadNodeW(childNodeID);
                    contextNode.ClearDataRowID(Key[index]);
                    return true;
                else if (childCount > 0)AND(i! = index) then
                    UpdateStatisticsInformation(KeyLength);
                    return true;
                else if (childCount == 0)AND(i! = index)AND(childNodeID == 0)
                then
                    UpdateStatisticsInformation(KeyLength);
                    return true;
                else
                    parentNodeID ⟵ contextNode.GetParentNodeID(Key[i])
                    contextNode.RemoveNode();
                    contextNode ⟵ ReadNodeW(parentNodeID);
                    contextNode ⟵ ClearChildID(Key[i]);
                childNodeID ⟵ contextNode.GetChildNodeID(Key[i]);
            end
            UpdateStatisticsInformation(KeyLength);
            return true;
        else if childNodeID == 0 then
            return false;
        else
            contextNode ⟵ ReadNodeW(childNodeID);
    end
End Function
```

Na začátku cyklu, je nutné znát počet potomků kontextového uzlu. Je-li tento počet větší, než 0 a jsme v první iteraci cyklu, pak stačí smazat ukazatele na hodnotu mazaného klíče a proces mazání je ukončen. Je-li počet potomků větší než 0, ale již se nenacházíme v první iteraci cyklu, pak je proces mazání opět úspěšně ukončen. Stejně tak, je-li počet potomků 0, nenacházíme se v první iteraci algoritmu a kontextový uzel je listový (*childNodeID=0*), pak je proces mazání úspěšně ukončen. V ostatních případech je nutné zjistit identifikátor předka kontextového uzlu pro danou položku, smazat kontextový uzel a nahradit jej rodičovským uzlem. Z rodičovského uzlu, tedy již z kontextového, je ještě nutné smazat ukazatel na uzel, který byl smazán. Na konec je nutné znát opět identifikátor potomka pro danou položku, abychom zachovali informaci o tom, co se stalo při předchozí iteraci. Cyklus končí, pokud je aktivní větev vracející *true*, nebo pokud cyklus doběhne do konce a je smazán i kořenový uzel. Před ukončením funkce je ještě nutné aktualizovat statistické informace o datové struktuře.

Třída *cTrieHeader* obsahuje konstruktor, který je nutné zavolat pro správné nastavení datové struktury. Tento konstruktor obsahuje několik parametrů 5.

Název parametru	**Popis parametru**	**Datový typ**
fileName	název souboru pro uložení hlavičky	char*
blockSize	velikost bloků cache bufferu	int
dimension	dimenze zpracovávaných záznamů	cDTDescriptor*
dataSize	velikost dat - hodnot záznamů	int
variableKey	záznamy proměnlivé délky	bool
seqNodeDepth	úroveň pro využití sekvenčních uzlů	int
seqNodeSize	velikost sekvenčního uzlu v prvcích	int
DSMode	mód datové struktury	int

Tabulka 5: Popis parametrů konstruktoru cTrieHeader

Poslední parametr určuje mód datové struktury. Tyto módy jsou zapsány ve třídě *cTrieConst*. Jedná se o mód *BASIC_TRIE*, který určí, že nebudou využívány sekvenční uzly, ale pouze asociační uzly. Dalším mód *SEQUENTIAL_NODE_TRIE* určuje, že sekvenční uzly jsou od určené hloubky Trie využívány.

4.2.1 Implementace položek Trie

Ke zpracování položek Trie slouží dvě třídy, které zajišťují správné načtení a uložení položek do jednotlivých typu uzlů Trie. Jedná se o třídy *cTrieItem* a *cTrieItemBitWorker*.

Třída *cTrieItemBitWorker* poskytuje metody pro výpočet identifikátoru pozice položky v bloku, dále budeme tento identifikátor nazývat *rowID*. Velikost *rowID* je dána šablonou, která určuje typ tohoto identifikátoru. Metoda *GetRowID* 5 přijímá jako parametr identifikátor bloku a pozici uzlu datové struktury v bloku, tedy identifikátor uzlu. Pomocí těchto parametrů se vypočítá *rowID* následujícím způsobem. Nejdříve je nutné vypočítat velikost posuvu identifikátoru

bloku, tuto velikost spočteme pomocí vztahu $log_2(velikost\_bloku)$. Pro velikost bloku 8192 B je vypočten posuv o 13 pozic vlevo. *RowID* je dále vypočteno jako logický součet posunutého identifikátoru bloku a identifikátoru uzlu. Třída *cTrieItemBitWorker* dále obsahuje metody pro zpětný výpočet identifikátoru bloku *GetBlockID* 6 a identifikátoru pozice uzlu *GetSubBlockID* 7. Obě tyto metody přijímají jako parametr *rowID*. Funkce *GetBlockID* provede bitový posuv *rowID* o $log_2(velikost\_bloku)$ pozic vpravo. Funkce *GetSubBlockID* provede logický součin *rowID* a velikosti bloku - 1. Výpočet logaritmů je ovšem drahá operace, proto je tato hodnota vypočtena při vytváření Trie a dále se nemění. Tato hodnota je stejně jako jiná metadata uložena v hlavičce datové struktury, tedy v instanci třídy *cTrieHeader*.

Algoritmus 5: Metoda GetRowID

Input: $blockID, offset$ - offset označuje pozici uzlu v datovém bloku
Output: $rowID$
Function `GetRowID(`$blockID, offset$`):`
  $temp \longleftarrow blockID << log_2(velikost\_bloku)$;
  return $offset|temp$;
End Function

Algoritmus 6: Metoda GetBlockID

Input: $rowID$
Output: $blockID$
Function `GetBlockID(`$rowID$`):`
  return $rowID >> log_2(velikost\_bloku)$;
End Function

Algoritmus 7: Metoda GetSubBlockID

Input: $rowID$
Output: $offset$
Function `GetSubBlockID(`$rowID$`):`
  return $rowID\&(velikost\_bloku - 1)$;
End Function

Třída *cTrieItem* obsahuje metody pro vložení a čtení položky v uzlu Trie. Dále obsahuje metodu *GetSize*, jejíž parametr určuje, zda jsou uloženy klíče pevné nebo pohyblivé délky. Pro klíče pevné délky je velikost položky v asociačním uzlu rovna velikosti *rowID*, protože to jsou jediná data, které je nutné znát pro přesměrování do dalšího uzlu Trie. Pro sekvenční uzel je tato velikost větší o velikost domény zpracovávaných záznamů, protože položky v sekvenčním uzlu nejsou uspořádány a je tedy nutné znát hodnotu konkrétní položky, protože není určena pořadím

v uzlu. Pro klíče pohyblivé délky obsahují položky Trie navíc další *rowID* stejné velikosti jako předešlé. Je to z toho důvodu, že u pohyblivé délky klíče může být konkrétní položka v uzlu listová a odkazovat se na hodnotu záznamu a současně může dále pokračovat jako klíč který má pouze společný prefix. Další metodou je *SetTrieItem* 8, která zapíše konkrétní položku na určené místo v uzlu. Další metody této třídy slouží k přečtení jednotlivých identifikátorů bloků a uzlů. Metoda *SetTrieItem* přijímá následující parametry 6.

Název parametru	**Popis parametru**	**Datový typ**
node	ukazatel na uzel datové struktury	&char*
blockID	identifikátor bloku pro identifikaci potomka	int
dataID	identifikátor datového bloku	int
dataOffset	identifikátor pozice v datovém bloku	int
variableKeyLength	záznam proměnlivé délky	bool
onlyData	vkládám pouze hodnotu záznamu	bool
nextSeq	uzel potomka je sekvenční	bool
nextSeq	uzel je listový	bool

Tabulka 6: Popis parametrů metody SetTrieItem

Metoda *SetTrieItem* 8 pracuje následovně. Nejdříve zjistíme, zda identifikátor bloku potomka není nula, tedy pokud existuje a současně nevkládám pouze data. Toto tedy znamená, že se nejedná o pouhou aktualizaci hodnoty záznamů. Následují podmínky podle jejichž splnění se vypočítá konkrétní identifikátor uzlu potomka (označeno jako *rowID*). První varianta, je-li potomek listovým uzlem a záznamy mají pevnou délku, pak je *rowID* určeno metodou *GetDataRowID*, protože vkládám do uzlu ukazatel na hodnotu záznamu. Další větev říká, že není-li uzel potomka sekvenční, použije se pro výpočet *rowID* metoda *GetRowID*. Třetí větev pak znamená, že uzel potomka je sekvenční a je tedy nutné použít metodu *GetSeqNodeRowID*. Po vypočtení *rowID* je tato adresa zkopírována do daného uzlu, tento uzel vstupuje do této metody už s požadovaným posunem, není zde tedy nutné určovat pozici uložení *rowID*. Poslední větev je navštívena vkládám-li ukazatele na hodnotu záznamu a zpracovávám záznamy proměnlivých délek. Jedná se tedy o aktualizaci hodnoty záznamu. V tomto případě je tedy nutné využívat více těchto identifikátorů, a proto je nutný posuv v rámci uzlu o velikost tohoto identifikátoru. Následně je ukazatel na hodnotu záznamu vložen do uzlu. Pokud dochází k aktualizaci hodnoty záznamů s pevnou délkou, je využita větev kde uzel je listový, tedy druhý *if* blok v pseudokódu.

4.2.2 Implementace uzlů Trie

Všechny uzly datové struktury Trie použité v této implementace, dědí ze třídy *cNode*. Jedná se o třídy *cTrieDataNode*, *cTrieAsociationNode* a *cTrieSequentialNode*.

Algoritmus 8: Metoda SetTrieItem

```
Input: node, blockID, offset, dataID, dataOffset, variableKeyLength, onlyData,
       nextSeq, isLeaf
Function SetTrieItem(node, blockID, offset, dataID, dataOffset,
                     variableKeyLength, onlyData, nextSeq, isLeaf):
    if blockID! = 0 AND !onlyData then
        if isLeaf AND !variableKeyLength then
            rowID ⟵ GetDataRowID(blockID, offset);
        else if !nextSeq then
            rowID ⟵ GetRowID(blockID, offset);
        else
            rowID ⟵ GetSeqRowID(blockID, offset);
        node ⟵ rowID;
    if dataID! = 0 AND variableKeyLength then
        (node + GetRowIDSize()) ⟵ GetDataRowID(dataID, dataOffset);
End Function
```

Třída *cTrieDataNode* reprezentuje uzel Trie, který slouží pouze k uložení hodnot záznamů. Data jsou do tohoto typu uzlu uložena sekvenčně a jednotlivé listové uzly Trie získají jejich pozici pomocí *rowID*. Třída obsahuje metody *AddData* 9 a *GetData* 10 s parametrem určující pozici v bloku. Metoda *AddData* dále přijímá jako parametr konkrétní data a jejich délku.

Metoda AddData 9 pouze inkrementuje proměnnou mRealSize, která určuje využitou veli-

Algoritmus 9: Metoda AddData

```
Input: offset, value, length
Function AddData(offset, value, length):
    mRealSize ⟵ length + mRealSize;
    fullOffset ⟵ GetTrieHeaderSize() + offset;
    index ⟵ 0;
    for i ⟵ fullOffset to fullOffset + length do
        mData[i] ⟵ value[index]
        index ⟵ index + 1
    end
End Function
```

kost bloku o délku hodnoty záznamů. Následně je vypočítán posuv v rámci uzlu, kam budou data zkopírována. Následuje cyklus, kde dochází k překopírování dané hodnoty záznamů na určené místo v datovém uzlu.

Metoda GetData 10 vrací data z datového uzlu z konkrétní pozice dané proměnnou offset.

Třída *cTrieAsociationNode* reprezentuje asociační uzly Trie. Tyto uzly jsou většinou menší, než velikost bloku a jsou proto do bloku uloženy sekvenčně. Třída obsahuje metodu *GetTrieElementarNode*, která vrátí konkrétní uzel Trie z bloku. Jako parametr přijímá tato funkce pouze

Algoritmus 10: Metoda GetData

Input: $offset$
Output: $Value$
Function `GetData(`$offset$`):`
  return $mData + offset + GetTrieHeaderSize()$;
End Function

pořadí hledaného uzlu v rámci bloku. Další metodou této třídy je *isPlaceForNewNode*, jak už název napovídá funkce vrací *true*, pokud je v bloku místo pro nový asociační uzel nebo *false* pokud už v bloku místo není. Poslední důležitou funkcí této třídy je metoda *AddItem*. Tato metoda zapíše pomocí metody *SetTrieItem* třídy *cTrieItem* vkládanou položku do uzlu, na pozici v uzlu, která je vrácena metodou *GetTrieElementarNode* s příslušným posuvem podle hodnoty konkrétní položky. Hodnotu posuvu určuje numerické vyjádření vkládané položky.

Třída *cTrieSequentialNode* obsahuje metody jako třída *cTrieAsociationNode*, navíc však obsahuje metody pro uložení nebo přečtení *rowID* následujícího sekvenčního uzlu, pokud existuje. Tyto metody jsou *GetNextRowID* pro přečtení a *AddNextRowID* pro zápis *rowID* následujícího sekvenčního uzlu. Velikost sekvenčního uzlu je zvolena při vytváření hlavičky Trie. Celková velikost uzlu v bytech je pak $velikost_itemu * zvoleny_pocet_item$ů$ + velikost_rowID$. Tato třída obsahuje dále metodu *GetTrieNodeOffset* 11, která vrátí pozici pro vložení položky do uzlu Trie, nachází-li se tato položka v uzlu, je vrácena její pozice.

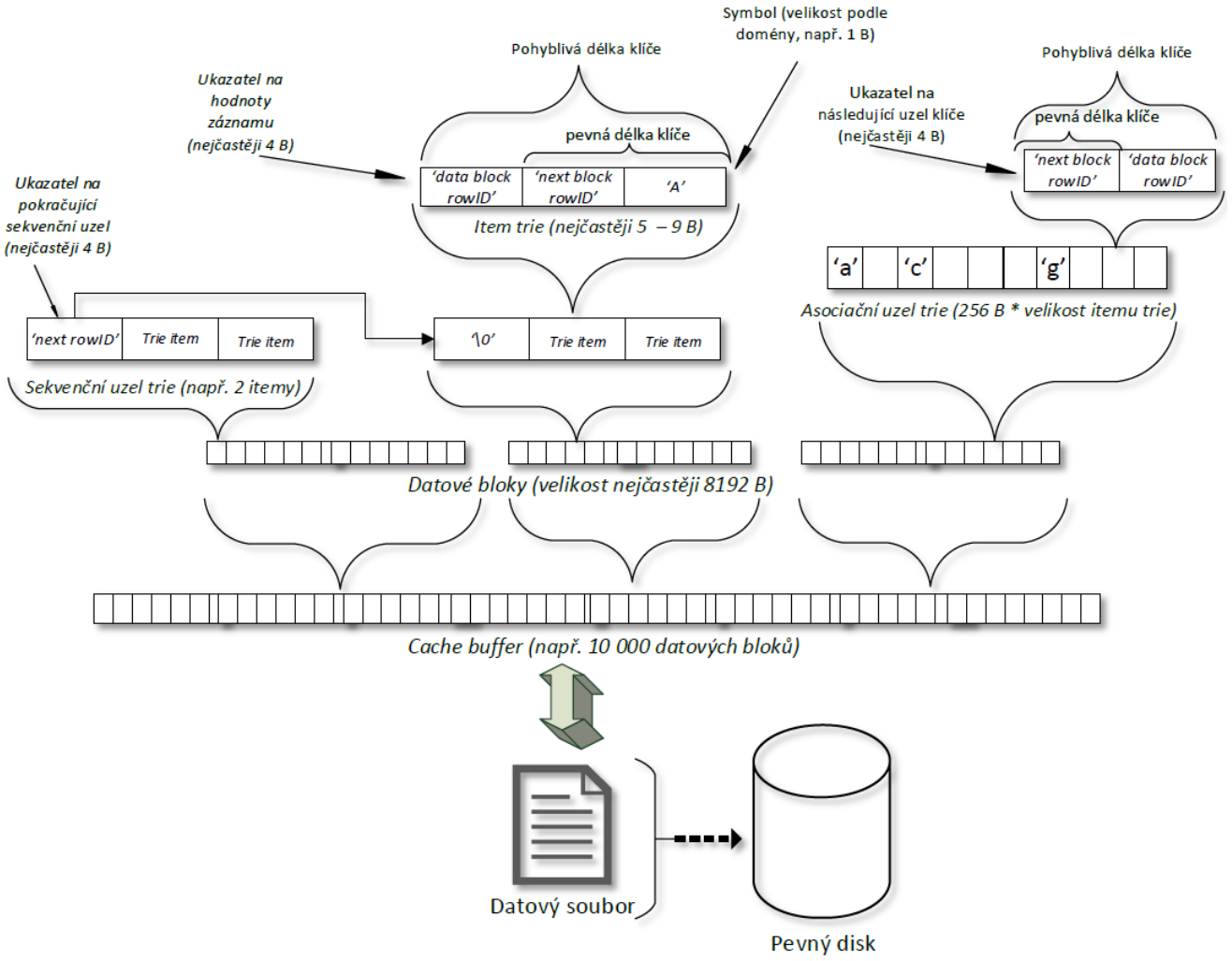
Vstupními parametry této metody jsou uzel, ve kterém probíhá hledání pozice a položka pro níž je tato pozice počítána. Metoda postupuje následovně. Nejdříve je vypočtena velikost prvku sekvenčního uzlu pomocí metody *GetSequentialItemSize*. Následně je vypočten posuv (*offset*), na místo, kde se nachází konkrétní hodnota, aby bylo možné ji porovnávat. Před vstupem do cyklu je ještě nutné znát počet položek v sekvenčním uzlu, tedy proměnnou *nodeItemsCount*. V cyklu procházíme jednotlivé položky a porovnáváme je s položkou, kterou hledáme v uzlu. Je-li tato položka v uzlu nalezena, vracíme začátek její pozice. Není-li nalezena, pak zjišťujeme, zda existuje sekvenční uzel zřetězený s tímto uzlem. Pokud ano, metoda vrací hodnotu 0, která identifikuje následný postup do dalšího sekvenčního uzlu. Nemá-li uzel žádného zřetězeného následovníka, pak je vrácena hodnota konce sekvenčního uzlu, která identifikuje, že se daná položka v sekvenčním uzlu nenachází.

Algoritmus 11: Metoda GetTrieNodeOffset

```
Input: node, item
Output: offset
Function GetTrieNodeOffset(offset):
    trieItemSize ⟵ GetSequentialItemSize();
    offset ⟵ GetRowIDSize() + (trieItemSize − GetDomainSize());
    nodeItemsCount ⟵ node + GetSequentialNodeSize() − GetItemsCountSize()));
    for i ⟵ 0 to nodeItemsCount do
        if item == (node + offset) then
            return offset − (trieItemSize − GetDomainSize());
        offset ⟵ trieItemSize + offset;
    end
    blockID ⟵ GetBlockID(node);
    if blockID! = 0 then
        return 0;
    else
        return offset − (trieItemSize − GetDomainSize());
End Function
```

Všechny tyto třídy reprezentující jednotlivé typy uzlů, obsahují navíc metody *Clear*, *Read* a *Write*. Metoda *Clear* slouží k inicializaci nového bloku, metoda *Read* slouží k načtení bloku ze sekundárního úložiště do cache bufferu a metoda *Write* pro uložení bloku z cache bufferu na sekundární úložiště.

Následující schéma 13 stručně ukazuje uložení jednotlivých položek v datovém souboru.



Obrázek 13: Schéma uložení položek Trie v datovém souboru

5 Experimenty

Jedním z hlavních cílů této diplomové práce, je otestování implementace Trie a srovnání výsledků těchto testů s testy jiných datových struktur jako B^+-strom a sekvenční pole, které jsou součástí databázového frameworku RadegastDB. Datové struktury byly otestovány jak z hlediska časové náročnosti operací (propustnosti), tak z hlediska paměťové náročnosti. Testování datových struktur jsem prováděl tak, aby byly všechny stránky datových struktur stále načteny v hlavní paměti. Velikost ukazatele na uzel, které budeme zkráceně říkat *hashID*, je během testování 4 B nebo 8 B. Maximální počet bloků identifikovatelných pomocí n-Bytů *hashID* je $2^{n*8-\lceil log_2(VelikostBloku[B])\rceil}$, tedy pro 4 B velikost domény 524 288 identifikovatelných bloků. Tam kde je během testování vyšší počet bloků, než zmíněných 2^{19}, tam je velikost *hashID* 8 B, v ostatních případech 4 B. Testování pro všechny datové kolekce bylo u všech testovaných struktur nastaveno následovně.

- Velikost Cache bufferu: 15 000 000
- Velikost bloku: 8192 B
- Délka dat: 8 B

5.1 Parametry testovacího hardwaru

Aplikaci jsem testoval na serveru dbedu.cs.vsb.cz s OS Windows Server 2016 a následujícími HW parametry.

- CPU Intel Xeon X5670, 24 logických jader, 2.93 GHz
- RAM 144 GB

5.2 Testování Trie

Velikost sekvenčních uzlů, včetně hloubky, v níž se začínají využívat, budou uvedeny před testem nad jednotlivými kolekcemi. Tyto hodnoty byly určeny na základě výkonnostních testů, který nejsou v této práci popsány, tak aby bylo možné propagovat co možná nejlepší konfiguraci Trie během testování. Trie využívající sekvenční uzly s 8-bitovou doménou (char) bude označována jako 8-Trie. Trie využívající sekvenční uzly s 16-bitovou velikostí domény (short) bude označena jako 16-Trie a Trie používající 32-bitovou doménu (integer) 32-Trie. Všechny tyto varianty testuji, abych poukázal na to, jak je nejvhodnější nastavit hloubku použití sekvenčních uzlů u jednotlivých velikostí domén Trie a porovnávám jejich paměťovou náročnost. Hloubka začátku využití sekvenčních uzlů se pro varianty Trie bude obvykle lišit, protože pro dosažení nejvyšších propustností struktury, se musí přizpůsobit dané datové kolekci.

Postup testování je následující. Jednotlivé datové kolekce budou načteny do pole záznamů s daným typem domény (vše v hlavní paměti), následně vloženy do datové struktury s variabilní

velikostí domény. Dotazování je prováděno na vzorcích, vygenerovaných z jednotlivých datových kolekcí a počet vzorků pro dotazování jsem stanovil na 100 000 záznamů. Ke generování vzorků pro dotazování je využita aplikace *CollectionGenerator*, která mi byla poskytnuta vedoucím této práce.

Během testování bude zaznamenávána propustnost operací vkládání a bodového dotazu v operacích za vteřinu. Dále bude zaznamenávána velikost datové struktury v MB, výška stromu (jedná-li se o stromovou datovou strukturu), počet využitých bloků a jejich průměrné využití v procentech. Dále bude u struktury Trie zaznamenáván počet jednotlivých typů uzlů. Testování datové struktury sekvenční pole, budu z důvodů šetření místa v tabulkách popisovat pouze jako 'Pole'.

5.3 Popis testovacích datových kolekcí

Datová kolekce	Počet záznamů	Dimenze dat	Typ domény
CARS[1]	3 360 277	4	unsigned int
CMS_STATS[2]	36 937 695	5	unsigned int
TIGER[3]	5 889 786	2	unsigned int
POKER[4]	1 000 000	11	char
DOMESTIC_FLIGHTS[5]	3 606 802	5	unsigned int
FORECAST[6]	98 947 177	7	unsigned int
GOOGLE_TRANSIT[7]	15 947 177	10	unsigned int
STOCKS[8]	19 610 502	11	unsigned int
KDDCUP[9]	909 108	42	unsigned int
HIGGS[10]	11 000 003	29	unsigned int
CLIMATOLOGY[11]	9 201 428	62	unsigned int

Tabulka 7: Testovací datové kolekce

Jednotlivé datové kolekce z tabulky 7 byly vybrány s důrazem na rozmanitost dimenzionality dat. Především tento aspekt bude hrát roli při následujícím testování. Datové kolekce obsahují minimálně přibližně 1 000 000 záznamů, aby bylo dosaženo co nejpřesnějšího testování. Byly

[1] http://www.census.gov/geo/www/tiger/

[2] http://www.cms.gov/Research-Statistics-Data-and-Systems/Files-for-Order/CostReports/Cost-Reports-by-Fiscal-Year.html

[3] http://www.census.gov/geo/www/tiger/

[4] http://archive.ics.uci.edu/ml/datasets/Poker+Hand

[5] http://www.infochimps.com/datasets/us-domestic-flights-from-1990-to-2009

[6] ftp://ftp.ncdc.noaa.gov/pub/data/ghcn/daily/grid/

[7] https://code.google.com/p/googletransitdatafeed/wiki/PublicFeeds

[8] http://www.infochimps.com/....

[9] http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

[10] https://archive.ics.uci.edu/ml/datasets/HIGGS

[11] https://data.nodc.noaa.gov/cgi-bin/iso?id=gov.noaa.ncdc:C00861

vybrány také datové kolekce s velmi vysokým počtem záznamů, řádově desítky miliónů, aby byla otestována struktura Trie i s 8 B *hashID*. Většina datových kolekcí má stejný typ domény, u Trie to ovšem téměř nehraje roli, protože je zde důležitá především velikost těchto typů domén. Většinou se tedy jedná o 4 B domény, nicméně dimenze dat je tak proměnlivá, že je Trie otestována pro nejrůznější délky řetězců.

5.3.1 Datová kolekce CARS

Nastavení sekvenčních uzlů pro tuto datovou kolekci je následující.

- Hloubka použití v 8-Trie: 6
- Hloubka použití v 16-Trie: 4
- Hloubka použití v 32-Trie: 3
- Velikost uzlu v položkách: 2

	8-Trie	**16-Trie**	**32-Trie**	**B$^+$-strom**	**Pole**
Propustnost vkládání [op/s]	1 332 000	1 342 000	530 260	1 151 784	21 000 723
Propustnost dotazování [op/s]	2 292 000	1 510 000	1 116 000	1 230 300	173
Počet využitých bloků	157 191	428 467	2 862 584	14 453	6 589
Průměrné využití bloku [%]	26.81	5.87	1.04	68.6	99.61
Velikost datové struktury [MB]	1 228.06	3 347.10	22 363.94	112.91	51.47
Výška stromu	16	11	10	2	-
Počet asociačních uzlů	620 246	2 742 836	8 503 098	-	-
Počet sekvenčních uzlů	33 649 697	15 454 952	6 079 633	-	-
Počet datových uzlů	3 286	3 286	3 286	-	-

Tabulka 8: Testování s datovou kolekcí CARS

Jak z testování v tabulce 8 vyplývá, je-li naším hlavním cílem dosáhnout nejvyšší propustnosti operací, pak je nejvhodnější použít Trie s 8-bitovou doménou, protože u ní lze nejpřesněji určit hloubku využívání sekvenčních uzlů Trie, díky čemuž dosáhneme u Trie nejlepšího využití bloků a současně nejmenší velikosti struktury. Navzdory vyšší výšce stromu než u ostatních velikostí domén Trie, je 8-Trie stále nejvhodnější k indexaci této datové kolekce. Pokud je ovšem hlavním cílem ušetřit velikost datové struktury, je nejvhodnější použít B$^+$-strom, jehož velikost je oproti 8-Trie téměř desetinásobně nižší, protože prostorová složitost Trie je mnohem větší viz kap 2.1.2 a současně propustnost vkládání jen o 200 000 op/s pomalejší, nevýhodou je ovšem propustnost dotazování, protože ta je oproti 8-Trie téměř poloviční. Lze tedy říci, že výběr vhodné struktury pro indexaci této datové kolekce záleží především na tom, zda upřednostňujeme vysokou propustnost nebo velikost struktury. Množství asociačních uzlů narůstá se zvyšující se velikostí domény,

protože jednotlivé typy Trie využívají různý počet úrovní asociačních uzlů, viz nastavení hloubky začátku využití sekvenčních uzlů. Ze stejného důvodů pak s rostoucí velikostí domény Trie klesá počet sekvenčních uzlů, protože se začínají využívat v nižších úrovních Trie (dál od kořene), dalším důvodem snižujícího se počtu sekvenčních uzlů je snižující se výška stromu, tedy klesá počet úrovní, ve kterých jsou využity sekvenční uzly. Počet datových uzlů je samozřejmě pro všechny velikosti domén stejný, protože délka dat, tedy hodnot z dvojice klíč-hodnota zůstává stejná. Počet datových uzlů bude pro jednotlivé typy Trie ve všech testech neměnný.

5.3.2 Datová kolekce TIGER

Nastavení sekvenčních uzlů pro tuto datovou kolekci je následující.

- Hloubka použití v 8-Trie: 4
- Hloubka použití v 16-Trie: 3
- Hloubka použití v 32-Trie: 2
- Velikost uzlu v položkách: 2

	8-Trie	**16-Trie**	**32-Trie**	**B$^+$-strom**	**Pole**
Propustnost vkládání [op/s]	1 897 000	1 028 603	1 040 000	1 122 262	37 042 679
Propustnost dotazování [op/s]	2 680 000	1 168 230	1 541 000	1 013 000	112
Počet využitých bloků	66 280	1 342 504	1 323 064	16 682	5 769
Průměrné využití bloku [%]	56.36	2.86	2.03	69.40	99.70
Velikost datové struktury [MB]	517.81	10 488.31	10 336.64	130.33	45.07
Výška stromu	8	6	5	2	-
Počet asociačních uzlů	65 792	3 894 134	3 894 134	-	-
Počet sekvenčních uzlů	26 074 556	10 488 147	4 603 108	-	-
Počet datových uzlů	5 753	5 753	5 753	-	-

Tabulka 9: Testování s datovou kolekcí TIGER

Výsledky testování v tabulce 9 dopadly následovně. Je-li naším cílem především vysoká propustnost operací, je pro indexaci této datové kolekce nejvhodnější využít 8-Trie, protože oproti vyšším velikostem domén Trie, lze u této Trie využít sekvenční uzly dříve, protože doména je 8-bitová, a záznam je tedy do každé úrovně Trie ukládán po 1B, proto lze úroveň využití sekvenčních uzlů nastavit vhodněji, a tím tedy snížit velikost datové struktury a současně lze díky tomu u této kolekce dosáhnout i vyšších propustností. 16-Trie a 32-Trie využívají sekvenční uzly od stejné hloubky (obě používají 4 úrovně asociačních uzlů, kdežto 8-Trie pouze 3), nicméně 32-Trie má vyšší propustnost, protože její výška je o jednu úroveň menší. Je-li naším cílem malá velikost struktury, pak je dle mého názoru stále nejvhodnější použít 8-Trie, protože její

velikost je pouze čtyřnásobná oproti B$^+$-stromu, ale propustnost vkládání je vyšší o téměř 70% a propustnost dotazování o přibližně 160%. Stejně jako u předešlého testování 8 roste počet asociačních uzlů se zvyšující se velikostí domény a ze stejných důvodů jako u předešlého testu 8 klesá počet sekvenčních uzlů. U 16-Trie a 32-Trie je počet asociačních uzlů stejný, protože využívají stejný počet úrovní asociačních uzlů.

5.3.3 Datová kolekce POKER

V této datové kolekci nelze využít Trie s vyšší velikostí domény, protože datová kolekce má znakový typ domény(1 B) a dimenze dat je lichá, nelze tedy využít sudou velikost domén. Nastavení sekvenčních uzlů pro tuto datovou kolekci je následující.

- Hloubka použití v 8-Trie: 5
- Velikost uzlu v položkách: 5

	8-Trie	**B$^+$-strom**	**Pole**
Propustnost vkládání [op/s]	1 220 118	659 207	15 384 555
Propustnost dotazování [op/s]	1 888 000	663 570	284
Počet využitých bloků	14 395	9 248	5 406
Průměrné využití bloku [%]	37.75	69.40	99.35
Velikost datové struktury [MB]	112.46	72.25	42.23
Výška stromu	11	2	-
Počet asociačních uzlů	264	-	-
Počet sekvenčních uzlů	3 518 089	-	-
Počet datových uzlů	979	-	-

Tabulka 10: Testování s datovou kolekcí POKER

Z tabulky 10 vyplývá, že pro indexaci této datové kolekce je opět nejvhodnější použít 8-Trie, navzdory tomu, že je oproti B$^+$-stromu o 50% větší, ale propustnost vkládání je téměř dvojnásobná a propustnost dotazování dokonce téměř trojnásobná. Z počtu asociačních uzlů lze vyčíst, že první 4 B záznamů z této datové kolekce bývá často stejných, protože je použito pouze 264 asociačních uzlů. Od 5. znaku záznamu se hodnoty začínají odlišovat, čímž narůstá i počet sekvenčních uzlů.

5.3.4 Datová kolekce DOMESTIC_FLIGHTS

Nastavení sekvenčních uzlů pro tuto datovou kolekci je následující.

- Hloubka použití v 8-Trie: 10
- Hloubka použití v 16-Trie: 6

- Hloubka použití v 32-Trie: 4
- Velikost uzlu v položkách: 3

	8-Trie	**16-Trie**	**32-Trie**	**B$^+$-strom**	**Pole**
Propustnost vkládání [op/s]	1 240 000	1 520 000	689 000	1 098 590	17 508 725
Propustnost dotazování [op/s]	2 008 000	2 337 000	1 347 000	1 116 000	158
Počet využitých bloků	108 520	224 692	2 290 076	18 546	8 841
Průměrné využití bloku [%]	37.97	10.94	1.23	67.10	99.60
Velikost datové struktury [MB]	847.81	1 755.41	17 891.22	144.89	69.07
Výška stromu	20	15	14	2	-
Počet asociačních uzlů	145 395	1 233 493	2 252 903	-	-
Počet sekvenčních uzlů	32 677 689	15 283 496	5 919 500	-	-
Počet datových uzlů	3 525	3 525	3 525	-	-

Tabulka 11: Testování s datovou kolekcí DOMESTIC_FLIGHTS

Z výsledků testování z tabulky 11 na této datové kolekci vyplývá, že nejlepších propustností je dosaženo u 16-Trie, protože oproti 8-Trie je výška stromu o 25% menší. U 32-Trie není dosaženo tak vysokých propustností, protože u 32-bitové domény nelze přesně určit hloubku kdy je nejvhodnější začít využívat sekvenční uzly. Je-li hlavním kritériem velikost datové struktury, potom je jednoznačně nejvýhodnější využít k indexaci této datové kolekce B$^+$-strom, protože jeho velikost je oproti 16-Trie více než desetinásobně menší. Počet asociačních uzlů opět narůstá s rostoucí velikostí domény jako u předchozích testů 8, stejně tak klesá počet sekvenčních uzlů ze stejných důvodů, jako v předešlých testech.

5.3.5 Datová kolekce CMS_STATS

Nastavení sekvenčních uzlů pro tuto datovou kolekci je následující.

- Hloubka použití v 8-Trie: 5
- Hloubka použití v 16-Trie: 3
- Hloubka použití v 32-Trie: 2
- Velikost uzlu v položkách: 2

	8-Trie	**16-Trie**	**32-Trie**	**B$^+$-strom**	**Pole**
Propustnost vkládání [op/s]	1 333 000	1 680 000	2 289 000	1 198 842	20 647 119
Propustnost dotazování [op/s]	2 582 000	3 173 000	3 652 000	953 212	13
Počet využitých bloků	1 290 599	423 286	254 486	246 810	90 534
Průměrné využití bloku [%]	60.31	59.21	63.14	51.70	99.61
Velikost datové struktury [MB]	10 082.81	3 306.92	1 988.17	1 928.17	707.30
Výška stromu	20	12	8	3	-

Tabulka 12: Testování s datovou kolekcí CMS_STATS

	8-Trie	**16-Trie**	**32-Trie**
Počet asociačních uzlů	194 697	194 697	194 697
Počet sekvenčních uzlů	346 118 914	162 757 367	70 657 375
Počet datových uzlů	36 109	36 109	36 109

Tabulka 13: Počet uzlů pro testování s datovou kolekcí CMS_STATS

Z výsledků testování v tabulce 12 vyplývá následující. Díky tomu, že nejvhodnější hloubka začátku využití sekvenčních bloků je 5, tedy ihned po prvních 4 B záznamu, lze u všech testovaných velikostí domén nastavit stejná hloubka použití sekvenčních uzlů, a tím docílit vysokých propustností s vhodnou paměťovou náročností. Je to způsobeno tím, že tato datová kolekce obsahuje v prvních 4 B záznamu hodnoty s nízkou četností. Díky tomu obsahují sekvenční uzly málo položek a prohledávání těchto uzlů je tedy rychlejší. K indexaci této datové kolekce je tedy jednoznačně nejvhodnější použít 32-Trie, protože dosahuje nejvyšších propustností a je pouze o 60 MB větší, než B$^+$-strom. Důvodem vyššího výkonu 32-Trie oproti 8-Trie a 16-Trie je to, že 32-Trie díky vyšší velikosti domény, má nejnižší výšku, což výrazně snižuje velikost 32-Trie, a tím snižuje počet logických přístupů během provádění testovaných operací. Tabulky 12 a 13 musely být z prostorových důvodů rozděleny. Počet asociačních uzlů je u všech typů Trie totožný, protože všechny tyto typy Trie využívají stejný počet úrovní asociačních uzlů. Počet sekvenčních uzlů opět klesá se zvyšující se velikostí domény ze stejných důvodů jako u testování předchozích datových kolekcí 8.

5.3.6 Datová kolekce FORECAST

Nastavení sekvenčních uzlů pro tuto datovou kolekci je následující.

- Hloubka použití v 8-Trie: 14
- Hloubka použití v 16-Trie: 8

- Hloubka použití v 32-Trie: 4
- Velikost uzlu v položkách: 2

Z výsledků testování v tabulce 14 vyplývá, že pro indexaci této datové kolekce je z výkonnostního hlediska nejvhodnější použít 16-Trie. Navzdory tomu, že 16-Trie nevyužívá sekvenční uzly od optimální hloubky, dosahuje vyšších propustností, než 8-Trie, protože má nižší výšku stromu. 16-Trie zabírá kvůli hloubce použití sekvenčních uzlů samozřejmě více paměti, než 8-Trie. 32-Trie je u této datové kolekce téměř nepoužitelná, protože nelze přesně určit začátek používání sekvenčních uzlů. Při výběru vyšší hloubky se datová struktura nevleze do paměti serveru, aniž by docházelo k přerozdělování paměti. Bylo tedy nutné použít nižší hloubku začátku použití sekvenčních uzlů, což sice snižuje velikost struktury i výšku stromu, ale současně dochází ke zřetězování sekvenčních uzlů, což zapříčiňuje výrazné zpomalení testovaných operací. Z hlediska paměťové náročnosti je pro tuto datovou kolekci nejvhodnější B^+-strom. Z těchto důvodů je i počet asociačních uzlů u 32-Trie mnohem nižší, než u ostatních velikostí domén viz 15. Opět zde klesá počet sekvenčních uzlů s rostoucí velikostí domény, důvod je opět stejný jako v předchozích případech 8.

	8-Trie	**16-Trie**	**32-Trie**	**B^+-strom**	**Pole**
Propustnost vkládání [op/s]	1 056 000	1 425 000	185 322	1 317 258	19 068 339
Propustnost dotazování [op/s]	2 255 000	2 950 000	256 387	617 284	2
Počet využitých bloků	3 176 576	3 541 921	878 202	669 814	340 025
Průměrné využití bloku [%]	63.57	31.11	73.02	65.70	99.46
Velikost datové struktury [MB]	24 818.22	27 670.12	6 860.95	5 232.92	2 656.45
Výška stromu	28	21	16	3	-

Tabulka 14: Testování s datovou kolekcí FORECAST

	8-Trie	**16-Trie**	**32-Trie**
Počet asociačních uzlů	93 124	5 830 369	70 644
Počet sekvenčních uzlů	887 201 755	406 988 090	180 481 063
Počet datových uzlů	96 662	96 662	96 662

Tabulka 15: Počet uzlů pro testování s datovou kolekcí FORECAST

5.3.7 Datová kolekce GOOGLE_TRANSIT

Nastavení sekvenčních uzlů pro tuto datovou kolekci je následující.

- Hloubka použití v 8-Trie: 6
- Hloubka použití v 16-Trie: 3
- Hloubka použití v 32-Trie: 2
- Velikost uzlu v položkách: 2

	8-Trie	**16-Trie**	**32-Trie**	**B$^+$-strom**	**Pole**
Propustnost vkládání [op/s]	558 238	883 687	1 389 000	1 017 625	22 163 225
Propustnost dotazování [op/s]	1 578 000	2 050 000	2 985 000	990 000	30
Počet využitých bloků	2 409 522	1 294 407	488 395	149 063	77 572
Průměrné využití bloku [%]	49.04	49.07	45.25	62.90	99.61
Velikost datové struktury [MB]	18 824.39	10 112.56	3 815.59	1 164.55	606.03
Výška stromu	40	22	13	3	-

Tabulka 16: Testování s datovou kolekcí GOOGLE_TRANSIT

	8-Trie	**16-Trie**	**32-Trie**
Počet asociačních uzlů	1 472 270	769 031	769 031
Počet sekvenčních uzlů	553 859 005	277 093 499	134 672 612
Počet datových uzlů	15 469	15 469	15 469

Tabulka 17: Počet uzlů pro testování s datovou kolekcí GOOGLE_TRANSIT

Výsledek testování 16 této datové kolekce dopadl podobně jako u kolekce CMS_STATS kap. 5.3.5, s tím rozdílem, že u 32-Trie a 16-Trie zde nelze nastavit hloubku začátku využití sekvenčních uzlů tak přesně, jako u 8-Trie, nicméně volbou vhodné blízké hloubce použití sekvenčních uzlů je u 16-Trie a 32-Trie dosaženo nejlepších propustností operací. Tím, že bylo zvoleno dřívější využití sekvenčních uzlů, bylo dosaženo radikálního snížení výšky stromu, což vede k nárůstu propustnosti operací Trie a současně ke zmenšení velikosti této struktury. 32-Trie je tedy nejvýkonnější testovaná struktura pro tuto datovou kolekci. Ovšem z hlediska velikosti struktur je nejvhodnějším kandidátem k indexaci této datové kolekce B$^+$-strom, jehož velikost je oproti 32-Trie zhruba třetinová. Z tabulky 17 je patrné, že počet asociačních uzlů opět roste s rostoucí doménou stejně jako v předchozích případech. U 16-Trie a 32-Trie je tento počet stejný, protože obě využívají 4 úrovně asociačních uzlů. Počet sekvenčních uzlů klesá s rostoucí velikostí domény stejně jako v předchozích případech viz 8.

5.3.8 Datová kolekce STOCKS

Nastavení sekvenčních uzlů pro tuto datovou kolekci je následující.

- Hloubka použití v 8-Trie: 9
- Hloubka použití v 16-Trie: 5
- Hloubka použití v 32-Trie: 3
- Velikost uzlu v položkách: 2

	8-Trie	**16-Trie**	**32-Trie**	**B$^+$-strom**	**Pole**
Propustnost vkládání [op/s]	525 108	845 814	1 290 000	1 156 741	17 052 603
Propustnost dotazování [op/s]	1 223 000	1 689 000	2 460 000	957 854	5
Počet využitých bloků	1 890 784	1 007 758	367 011	250 755	106 003
Průměrné využití bloku [%]	61.21	61.24	59.09	50.40	99.36
Velikost datové struktury [MB]	14 771.75	7 873.11	2 867.27	1 959.02	828.15
Výška stromu	44	26	17	3	-

Tabulka 18: Testování s datovou kolekcí STOCKS

	8-Trie	**16-Trie**	**32-Trie**
Počet asociačních uzlů	14 654	14 654	14 654
Počet sekvenčních uzlů	543 177 135	266 553 703	128 241 987
Počet datových uzlů	19 170	19 170	19 170

Tabulka 19: Testování s datovou kolekcí STOCKS

Výsledek testování 18 této datové kolekce dopadl podobně jako u kolekce CMS_STATS kap. 5.3.5. I zde je nejvhodnější hloubka pro začátek využívání sekvenčních uzlů stejná pro všechny typy Trie, což zapříčiňuje s rostoucí velikostí domény snižování výšky stromu, a současně rostoucí propustnost operací i klesající velikost této struktury. Pro indexaci této datové struktury je tedy nejvhodnější využít 32-Trie, protože dosahuje nejvyšších propustností a oproti B$^+$-stromu je jen o přibližně 1 GB větší. V tabulce 19 vidíme že, počet asociačních uzlů je u všech typů Trie stejný, protože všechny využívají 8 úrovní asociačních uzlů. Počet sekvenčních uzlů klesá s rostoucí velikostí domény stejně jako v předchozích případech 8.

5.3.9 Datová kolekce KDDCUP

Nastavení sekvenčních uzlů pro tuto datovou kolekci je následující.

- Hloubka použití v 8-Trie: 6
- Hloubka použití v 16-Trie: 6
- Hloubka použití v 32-Trie: 11
- Velikost uzlu v položkách: 2

	8-Trie	**16-Trie**	**32-Trie**	**B$^+$-strom**	**Pole**
Propustnost vkládání [op/s]	109 308	120 000	189 340	602 383	10 101 116
Propustnost dotazování [op/s]	225 870	289 013	405 890	652 646	154
Počet využitých bloků	2 344 057	2 218 241	2 154 926	29 515	18 940
Průměrné využití bloku [%]	7.29	4.22	2.56	69.00	98.44
Velikost datové struktury [MB]	18 312.92	17 330.01	16 835.36	230.59	147.97
Výška stromu	168	104	72	3	-

Tabulka 20: Testování s datovou kolekcí KDDCUP

	8-Trie	**16-Trie**	**32-Trie**
Počet asociačních uzlů	29 905	85 714	6 219 019
Počet sekvenčních uzlů	84 939 375	42 267 241	19 365 120
Počet datových uzlů	889	889	889

Tabulka 21: Počet uzlů pro testování s datovou kolekcí KDDCUP

Výsledek testování 20 dopadl následovně. Jelikož má tato datová kolekce vysokou dimenzi dat, je pro tuto datovou kolekci nevhodné využívat Trie, protože s rostoucí délkou záznamu roste i výška Trie, a tím klesá propustnost operací a současně roste její velikost. Pro indexaci této kolekce je tedy nejvhodnějším řešením B$^+$-strom. Počet uzlů z tabulky 21 se chová jinak než v předešlých případech. Počet asociačních uzlů zde roste s rostoucí velikostí domény. To je zapříčiněno tím, že dle nastavení hloubky začátku využívání sekvenčních uzlů používáme u 8-Trie 5 úrovní asociačních uzlů. U 16-Trie je to pak 10 úrovní a u 32-Trie dokonce 40 úrovní. To je tedy důvodem nárůstu množství asociačních uzlů. Množství sekvenčních uzlů zde naopak klesá, protože s rostoucím počtem úrovní asociačních uzlů, klesá počet úrovní využití sekvenčních uzlů. Tento jev pouze nepatrně vyrovnává snižování výšky stromu.

5.3.10 Datová kolekce HIGGS

Nastavení sekvenčních uzlů pro tuto datovou kolekci je následující.

- Hloubka použití v 8-Trie: 21
- Hloubka použití v 16-Trie: 11
- Hloubka použití v 32-Trie: 6
- Velikost uzlu v položkách: 2

	8-Trie	**16-Trie**	**32-Trie**	**B$^+$-strom**	**Pole**
Propustnost vkládání [op/s]	205 000	295 054	395 314	342 658	12 614 675
Propustnost dotazování [op/s]	306 000	448 823	515 214	396 353	4
Počet využitých bloků	1 529 121	489 517	292 223	229 941	157 143
Průměrné využití bloku [%]	61.15	57.28	58.38	75.00	99.12
Velikost datové struktury [MB]	11 946.26	3 824.35	2 282.99	1 796.41	1 227.68
Výška stromu	116	68	44	4	-

Tabulka 22: Testování s datovou kolekcí HIGGS

	8-Trie	**16-Trie**	**32-Trie**
Počet asociačních uzlů	4 277	4 277	4 277
Počet sekvenčních uzlů	441 435 691	216 617 655	104 208 637
Počet datových uzlů	10 710	10 710	10 710

Tabulka 23: Počet uzlů pro testování s datovou kolekcí HIGGS

Z výsledků testování 22 vyplývá, že nejvhodnější nastavení hloubky začátku využití sekvenčních uzlů je u všech typů Trie stejné, čímž lze dosáhnout pro jednotlivé typy nejlepších výsledků. Opět je zde vidět, že s klesající výškou stromu klesá i velikost Trie a současně roste její propustnost. Propustnost vkládání je u 32-Trie asi o 50 000 vložení za vteřinu rychlejší, než v případě B$^+$-stromu a u dotazování je to o přibližně 100 000 dotazů za vteřinu. Velikost 32-Trie je však o 27% větší, než B$^+$-strom. Zde je poměrně složité jednoznačně rozhodnout, která struktura je pro indexaci této datové kolekce nejvhodnější. Upřednostňujeme-li propustnost operací zvolíme 32-Trie, upřednostňujeme-li velikost indexu, pak zvolíme B$^+$-strom. Z tabulky 23 je patrné, že počet asociačních uzlů je pro všechny velikosti domén Trie shodný, protože všechny využívají 20 úrovní asociačních uzlů. Tato datová kolekce obsahuje velké množství záznamů s opakujícími se prvními 20 B, protože při 20 úrovních asociačních uzlů jich obsahuje pouze 4 277. Z důvodů snižování výšky stromu s rostoucí velikostí domény, klesá počet sekvenčních uzlů.

5.3.11 Datová kolekce CLIMATOLOGY

Nastavení sekvenčních uzlů pro tuto datovou kolekci je následující.

- Hloubka použití v 8-Trie: 29
- Hloubka použití v 16-Trie: 15
- Hloubka použití v 32-Trie: 8
- Velikost uzlu v položkách: 2

	8-Trie	**16-Trie**	**32-Trie**	**B$^+$-strom**	**Pole**
Propustnost vkládání [op/s]	98 859	129 703	164 628	412 353	7 349 378
Propustnost dotazování [op/s]	166 524	198 117	262 547	382 409	11
Počet využitých bloků	10 641 791	8 481 430	7 395 732	328 143	287 545
Průměrné využití bloku [%]	26.69	17.87	11.53	94.70	96.87
Velikost datové struktury [MB]	83 138.99	66 261.14	57 7779.16	2 563.62	2 246.45
Výška stromu	248	138	83	4	-

Tabulka 24: Testování s datovou kolekcí CLIMATOLOGY

	8-Trie	**16-Trie**	**32-Trie**
Počet asociačních uzlů	17 935 392	17 935 392	17 935 392
Počet sekvenčních uzlů	1 354 393 162	675 849 149	336 561 377
Počet datových uzlů	9 040	9 040	9 040

Tabulka 25: Počet uzlů pro testování s datovou kolekcí CLIMATOLOGY

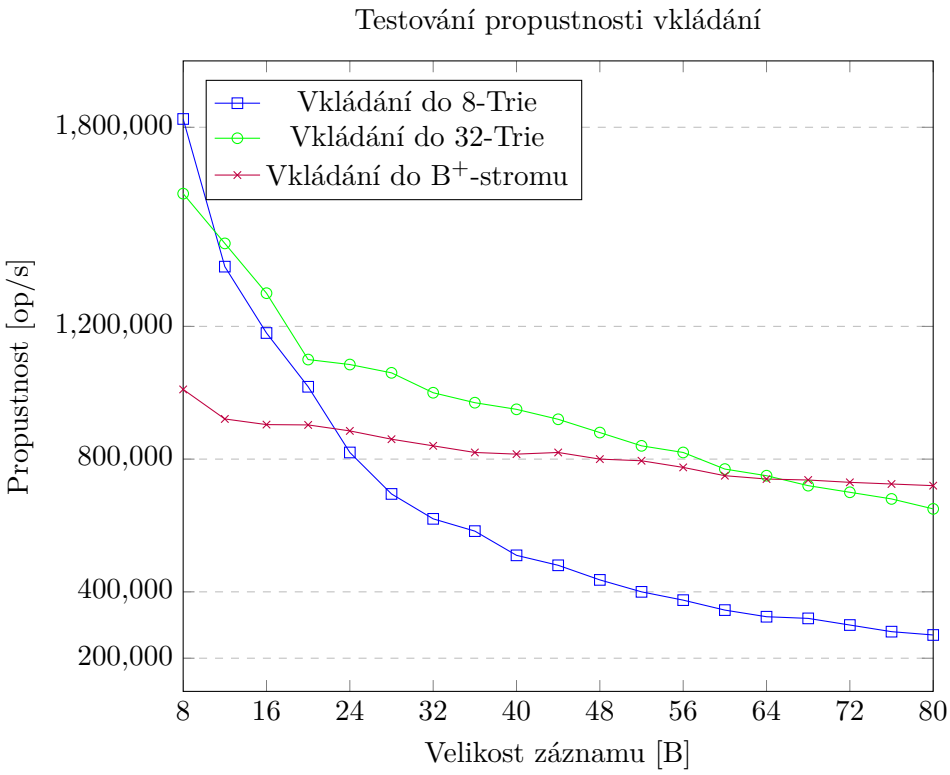
Z výsledků testování 24 vyplývá, že kvůli vysoké dimenzi dat a s tím související výšce stromu je Trie pro indexaci této kolekce naprosto nevhodná, protože dosahuje nízkých propustností a obrovské velikosti. Zde se ukazuje hlavní nevýhoda Trie, jak už vyplývá z časové složitosti operací Trie, s rostoucí délkou klíč roste i čas potřebný k provedení operace. Současně také roste její velikost. Z těchto důvodů je pro indexaci této datové kolekce jednoznačně nejvhodnější B$^+$-strom. Z tabulky 25 je patrné, že počet asociačních uzlů je pro všechny velikosti domén stejný, to je opět zapříčiněno tím, že všechny typy Trie využívají 28 úrovní asociačních uzlů. Počet sekvenčních uzlů opět klesá s rostoucí velikostí domény, protože s rostoucí velikostí domény rovněž klesá výška stromu.

5.4 Závislostní testování

Datová struktura Trie se ukazuje jako nevhodná pro indexaci záznamů s vysokou dimenzi dat. V následujících testech ukáži, jak závisí velikost záznamu (dimenze dat) na propustnosti operací Trie a na její velikosti. Jednotlivé záznamy určené k testování mají normální rozložení a jsou generovány pomocí instance třídy *cTuplesGenerator*, která je součástí RadegastDB frameworku. Hloubka začátku použití sekvenčních uzlů je pro každý typ Trie rozdílná, aby bylo dosaženo co nejvyšších propustností. Nastavení testů a Trie je následující:

- Počet testovaných záznamů: 1 000 000
- Velikost Cache bufferu: 500 000 bloků
- Velikost bloku: 8192 B
- Velikost sekvenčního uzlu Trie v položkách: 3
- Hloubka použití sekvenčních uzlů v 8-Trie: 4
- Hloubka použití sekvenčních uzlů v 32-Trie: 2

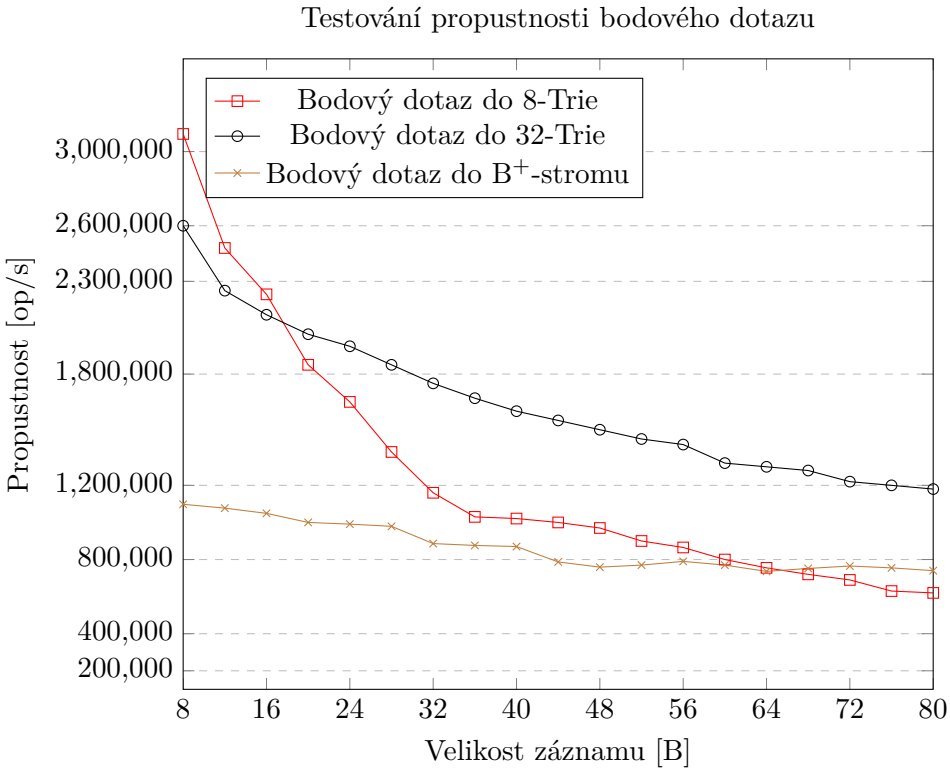
5.4.1 Testování závislosti propustnosti operací Trie na velikosti záznamu



Obrázek 14: Graf závislosti propustnosti vkládání Trie na velikosti záznamu

Z grafu 14 je vidět, že vkládání do 8-Trie je pro nízkou dimenzi dat nejvhodnější, protože lze přesněji určit hloubku začátku využití sekvenčních uzlů. Nicméně s rostoucí velikostí záznamu je vhodnější použít větší velikost domény pro Trie, protože je tím redukována výška stromu, což zapříčiňuje vyšší propustnost vkládání. Propustnost vkládání u 32-Trie neklesá tak prudce, jako 8-Trie. 32-Trie je ke vkládání nejvhodnější nepřesahuje-li velikost záznamu 64 B. Od vyšší velikosti záznamů je rychlejší B$^+$-strom, protože časová složitost jeho operací je logaritmická a nezávisí na velikosti záznamu, nicméně je vidět, že i u B$^+$-stromu propustnost vkládání klesá, protože s rostoucí velikostí záznamu roste i počet porovnávání nutných ke správnému zařazení záznamu do B$^+$-stromu.

V grafu 15 lze vidět, že bodový dotaz do 8-Trie dosahuje nejvyšších propustností, protože u



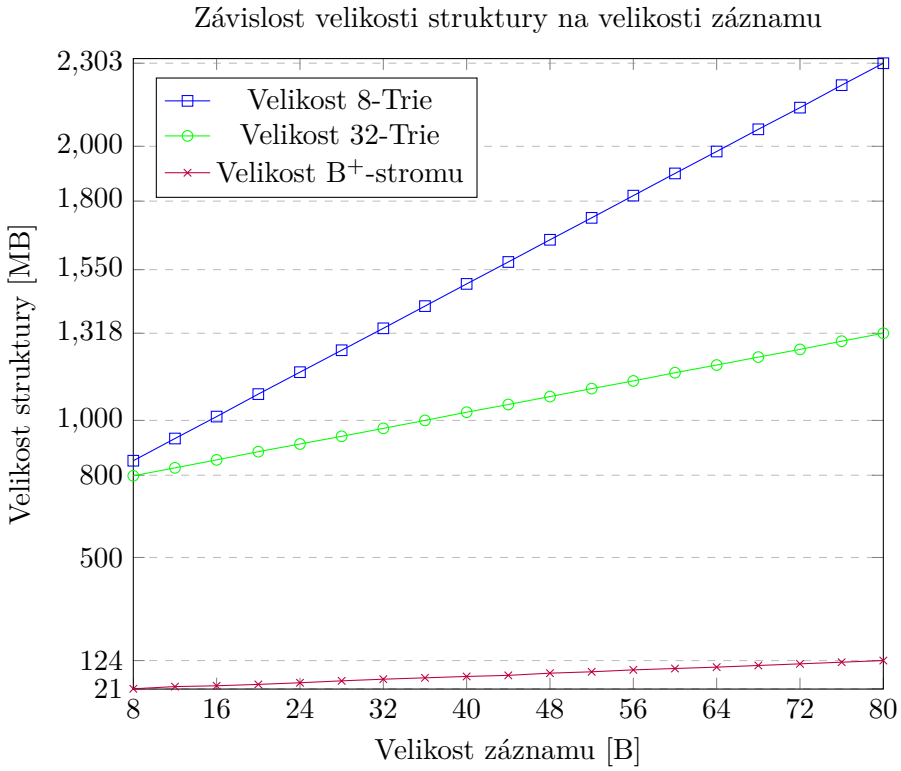
Obrázek 15: Graf závislosti propustnosti dotazování Trie na velikosti záznamu

8-Trie lze nejpřesněji nastavit hloubka začátku využití sekvenčních uzlů. Od velikosti záznamu 20 B dosahuje nejvyšších propustností 32-Trie, protože čtyřnásobná velikost domény Trie značně zredukuje výšku stromu, což vede k lepší propustnosti operací. Propustnost dotazování u 32-Trie neklesá tak rychle, jako u 8-Trie, protože výška 8-Trie roste rychleji. Propustnost dotazování u B$^+$-stromu opět pozvolna klesá, protože s rostoucí dimenzi dat roste i počet porovnávání pro nalezení záznamu v B$^+$-stromu, tato propustnost ovšem neklesá tak rychle, jako u 32-Trie, takže lze očekávat, že někde za hranicí grafu bude B$^+$-strom opět dosahovat vyšší propustnosti dotazování, než 32-Trie.

5.4.2 Testování závislosti velikosti Trie na velikosti záznamu

Pro testování velikosti jednotlivých typů Trie během rostoucí velikosti záznamu, jsem nastavil hloubku začátku používání sekvenčních uzlů na stejnou hodnotu, aby bylo dosaženo co možná nejsrovnatelnějších výsledků. Sekvenční uzly byly přenastaveny následovně.

- Hloubka použití sekvenčních uzlů v 8-Trie: 5
- Hloubka použití sekvenčních uzlů v 32-Trie: 2



Obrázek 16: Graf závislosti velikosti Trie na velikosti záznamu

Z grafu 16 vyplývá, že, velikosti typů Trie začínají u přibližně stejné hodnoty. Velikost 8-Trie však stoupá mnohem strměji, než 32-Trie, protože při každém nárůstu velikosti záznamu se zvýší výška stromu o hodnotu rozdílu mezi měřenými velikostmi. U 32-Trie se s každým zvětšením velikosti záznamu zvýší výška pouze o 1, protože velikost záznamu se zvyšuje o 4 B a doména 32-Trie je stejné velikosti. Velikost B^+-stromu je však nesrovnatelně menší, protože B^+-strom dosahuje lepšího využití jednotlivých uzlů a současně využívá menší počet uzlů. Nárůst velikosti B^+-stromu je také menší, protože výška stromu není přímo závislá na velikosti záznamu.

6 Závěr

Tato práce si kladla za cíl navrhnout a naimplementovat datovou strukturu Trie v jazyce *C++*, včetně jejího rozšíření o sekvenční uzly, nastavitelné velikosti domény a zakomponování této struktury do databázového frameworku RadegastDB. Všech těchto cílů bylo dosaženo a byla naimplementována datová struktura Trie, splňující tyto požadavky. Dalším cílem této práce bylo implementaci řádně otestovat a porovnat s dalšími strukturami.

Trie se ukázala vhodná pro indexaci datových kolekcí s nízkou dimenzí dat, přibližně do 20 s 4 B velikostí domény. Různé velikosti domény Trie dosahují různých výsledků, nicméně se ukazuje, že pro nejmenší záznamy, přibližně do 16 B je nejvhodnější využít Trie s 8-bitovou doménou, protože dosahovala nejlepšího výkonu mezi testovanými strukturami. Pro větší záznamy je pak nejvhodnější Trie s 32-bitovou doménou, jejíž výška stromu je mnohem nižší, než u menších velikostí domén Trie a díky tomu dosahuje lepších výkonů. Přibližně od 100 B záznamů je struktura Trie nevhodná, protože již nedosahuje vyšších propustností, než např. B^+-strom. Velkou nevýhodou Trie je její velikost, u všech podstoupených testování byla několikanásobně větší, než zmíněný B^+-strom a nepopsatelně větší než sekvenční pole. Z tohoto důvodu je nutné vždy zvažovat, zda použijeme k indexaci dané datové kolekce strukturu Trie, protože se může snadno stát, že její velikost bude mimo technické možnosti datových úložišť.

Tato práce se snažila poskytnout detailní popis datové struktury Trie, včetně její optimalizace. Nejsložitějším aspektem této práce bylo zakomponovat sekvenční uzly do algoritmu vkládání do Trie, dále zajistit správné stránkování struktury tak, aby nedocházelo ke ztrátě informací během zápisu na sekundární úložiště.

Dalším vhodným rozšířením této práce je například implementace dalších typů uzlů, či dynamický přechod mezi použitím asociačních a sekvenčních uzlů, což by mohlo částečně vyřešit problém s velikostí této struktury. Jiným vhodným rozšířením, nebo spíše modifikací je implementace zmíněných variant Trie, jako sufixový strom, nebo komprimovaná Trie, které taktéž snižují paměťovou náročnost Trie, ovšem na úkor propustnosti při modifikaci struktury.

Literatura

[1] Peter Brass, *Advanced Data Structures*, City college of New York 2008, pages 336-356.

[2] R. de la Briandais, *File searching using variable-length keys* In Proceedings of the Western Joint Computer Conference, pages 295–298.

[3] E. Fredkin, *Trie memory* Communications of the ACM, pages 490–499.

[4] Arnab Bhattacharya, *Fundamentals of Database Indexing and Searching* Indian Institute of Technology (IIT), Kanpur India, pages 51-54 and 217.

[5] Dinesh P. Mehta, Sartaj Sahni, *Handbook of DATA STRUCTURES and APPLICATIONS* Chapman & Hall/CRC computer & information science, pages 561-572.

[6] Hanan Samet, *Foundations of Multidimensional and Metric Data Structures*, University of Maryland, College Park, 2006.

[7] Radim Bača, Peter Chovanec, Michal Krátký, Petr Lukáš *QuickDB – Yet Another Database Management System?*, Department of Computer Science, FEECS, VSB – Technical University of Ostrava 2015.

[8] Ing. Peter Chovanec, Ph.D. *REDUCTION OF DISK ACCESSES IN MULTIDIMENSIONAL DATA STRUCTURES*, Dizertační práce VŠB – Technical University of Ostrava Faculty of Electrical Engineering and Computer Science Department of Computer Science, Ostrava 2015, pages 19-20.

[9] T. Lahdenmäki and M. Leach. *Relational Database Index Design and the Optimizers*, John Wiley and Sons, New Jersey, 2005.

[10] Kurt Maly. *Compressed tries*, Communications of the ACM, 1976, page 53.

[11] D. Shasha and P. Bonnet. *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*, Morgan Kaufmann, 2002.

[12] Elizabeth J. O'Neil, Patrick E. O'Neil, Gerhard Weikum. *The LRU-K page replacement algorithm for database disk buffering*, 1993.

[13] Ulrich Germann, Eric Joanis, Samuel Larkin. *Tightly packed tries: how to fit large models into memory, and make them load fast, too*, University of Toronto and National Research Council Canada, 2009.

[14] Gaston H. Gonnet. *Unstructured Data Bases or Very Efficient Text Searching*, Department of Computer science, University of Waterloo, 1983.

A Příloha na CD

\RadegastDB\src	Adresář frameworku RadegastDB
\RadegastDB\src\common	Zdrojové kódy obslužných tříd frameworku RadegastDB
\RadegastDB\src\common\datatype	Zdrojové kódy datových typů frameworku RadegastDB
\RadegastDB\src\dstruct	Zdrojové kódy datových struktur frameworku RadegastDB
\RadegastDB\src\dstruct\paged\core	Zdrojové kódy jádra frameworku RadegastDB
\RadegastDB\src\dstruct\paged\trie	Zdrojové kódy datové struktury Trie
\RadegastDB\src\dstruct\paged\b+tree	Zdrojové kódy datové struktury B^+-strom
\RadegastDB\src\dstruct\paged\sequentialarray	Zdrojové kódy datové struktury sekvenční pole
\RadegastDB\src\test\paged\trie_test	Testovací aplikace datové struktury Trie
\RadegastDB\src\test\paged\btree_test	Testovací aplikace datové struktury B^+-strom
\RadegastDB\src\test\paged\sequentialarray_test	Testovací aplikace datové struktury sekvenční pole

Tabulka 26: Obsah CD